Mitigating the effects of vendor lock in in edge cloud environments with open source technologies

Gábor Finta

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Stockholm 25.09.2019

Supervisor

Prof. Jukka K. Nurminen

Advisor

Mihhail Matskin

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Gábor Finta

Title Mitigating the effects of vendor lock in in edge cloud environments with open source technologies

Degree programme School of Science

Major Cloud Computing and Services **Code of major** SCI3081

Supervisor Prof. Jukka K. Nurminen

Advisor Mihhail Matskin

Date 25.09.2019 **Number of pages** 56 **Language** English

Abstract

Cloud computing has been in the center of attention recently. Its popularity has increased significantly. More and more companies decide to use a cloud for running their applications. However, this introduces certain problems, such as vendor lock-in. Without a widely used standard, the systems become incompatible with each other. This thesis introduces a way to reduce the risk of vendor lock-in and uses open source technologies in order to make it available to as many people as possible. The explored solution is easy-to-use and light-weight compared to other ones. Furthermore, the use of certain technologies over others is suggested in the thesis to further reduce the risks of being locked to a single cloud provider.

Keywords cloud, lock-in, compatibility, portability

Contents

1 Introduction

This chapter presents the basis of the project as well as the problem, research question and goals.

1.1 Background

1.1.1 Cloud Computing

Cloud computing has gained a significant amount of attention recently, and it became widely used [1]. This is mostly due to the benefits it offers. One major advantage for the organizations using these services is that there is no need for an initial investment in servers. Instead of buying machines, installing the required software and managing the system, hosts in the cloud can be used with much less effort and initial money. Another great benefit is on the side of the cloud providers. Managing large scale data centers is more cost-efficient than multiple smaller ones. Most of the advantages of cloud computing are due to the fact that the data centers are centralized and large scale entities. One important characteristic of such a paradigm is rapid elasticity [2]. In a hypothetical scenario, where we have a web server running, and suddenly the traffic soars, multiple instances of new replicas of the server can be spun up to distribute traffic. Latency difference should be negligible as all the machines are in the same data center [3].

Centralization comes with many advantages but it fails to fulfill some emerging needs. In general, data centers are relatively far away from the end-user and this comes with higher latency. Some latency-critical applications, such as the ones using augmented reality, require it to be as low as possible. This requirement introduced the need for edges. The principal idea of edges is to take the computation closer to the data [4]. This can reduce the latency of the applications significantly. In some cases, the edges are used to preprocess the raw data and send the result of this to the cloud for further processing. This can be useful when the preprocessing can significantly reduce the size of the data while the rest of the task is computationally heavy and must be transferred to the central cloud. Edges are becoming more and more relevant as the hardware get more powerful [5]. Lower latency enables latency-critical applications to thrive, such as real-time image recognition. Another example is Google's Stadia [6], which is aiming to change one of the biggest industries, the gaming industry. The user only needs a display because the game itself runs on Google computers. With a latency low enough for gaming, it can have a huge impact on this industry.

The current trend is to migrate to the cloud. However, this comes with certain risks. Organizations that decide to migrate their applications can face the problems of vendor lock-in [7]. Different cloud providers can use different technologies to provide their services and they tend to be incompatible with each other. This comes with several consequences and problematic scenarios:

- The cloud provider in use raises its prices or another one offers discounts.
- In case of an outage, a certain provider can be unavailable for use.
- The data gets stuck in the database and the organization is unable to move it somewhere else.

With the rise of cloud computing, relevant technologies emerged as well. In this thesis, the focus is put to open source projects because they have the potential to be widely used, thus having a higher impact.

1.1.2 Open Source

Open source software development has seen an upward trend lately. Numerous new open source projects have been started and many closed source projects have been made open source. There are at least two points of views one should consider to understand the reason behind this trend. The developers can have multiple different reasons to contribute to these projects [8]:

- They are paid for it by a company to improve the software.
- They simply enjoy contributing to complex projects.
- It can be a reference for future employment purposes.

From a company's point of view, making their own closed source project open source can attract developers or other companies from all over the world to improve the software more. Bugs or security issues can be detected more effectively and new feature ideas can come from a more diverse group of developers. These result in a better project that companies can build a service on and convert this into revenue.

This thesis focuses on open source projects as they are more accessible for users and developers than paid software in the sense that even small companies or individuals with less resources than big companies can afford to use these tools. It is important to target a user base that has the potential to make a real difference in the cloud industry.

1.2 Problem statement

Cloud providers offer numerous tools for a user to host applications, databases and so on. However, even though these services are comparable, they are usually incompatible with each other. This means that if an application is developed for the platform of one particular cloud provider, it is likely that the whole application needs to be rewritten if migrated to another provider. The same problem arises with data storages, such as different databases, which further complicates the situation [7]. Vendor lock-in can have multiple consequences:

- The cloud provider in use increases prices but the user is stuck with it even if saving money would be possible with other providers.

- The provider has a service outage, which still happens [9], and the user can only wait for the service to come back up.
- The user would like to move to an on-premise cloud for cost reduction or privacy purposes but is stuck with the current provider.

These consequences have serious on the cost or high availability of a service. Cloud providers tend to follow no standards but the one they designed themselves [7]. The burden of handling the problem falls on the users. Finding available tools or implementing new ones that can mitigate the impact of vendor lock-in is an emerging issue.

1.3 Purpose

The purpose of the thesis is to tackle the problem of vendor lock-in and reduce the risk of this happening. Cloud computing users should be aware of this problem and have tools for solving them.

1.4 Goal

The primary goal of the thesis is to show the users exact ways or tools that can be used to solve their problems caused by vendor lock-in. Furthermore, it is necessary that the provided tools are proven to be capable of tackling this problem.

1.5 Ethics and sustainability

With the increasing trend of moving to the cloud and using high-performance computers offered there, energy consumption increased as well [10]. While vendor lock-in keeps some companies away from clouds [7], the reduced risk of such a problem would attract them towards it and more instances and applications could be migrated to the cloud using high-performance computers. This increases electricity usage even more.

Keeping user data in a public cloud instead of a company's own server machines has higher privacy concerns. Migration in between multiple public clouds further increases the risks of a breach as more systems can have more weak points.

1.6 Methodology

The methodology applied in this thesis is empirical experimenting and testing. The data is collected by conducting experiments, then it is analyzed and the results are concluded. The thesis can be broken down into the following steps:

1. Analyzing and studying the background of the problem.
2. Comparing and analyzing tools in this problem area.

3. Designing the test scenario for the experiments.
4. Setting up the work and experimental environment where the tests are conducted.
5. Experimenting with the tools and applications to gather data.
6. Concluding the results and discussing possible future work.

1.7 Delimitations

There are multiple software projects that can be used to build up an infrastructure and run containerized applications but this thesis focuses on open source technologies as it has the highest potential to reach the most people. Kubernetes is used in the experiment environment since it is the most popular container orchestration tool, currently. Thus, the thesis only provides value to the users of Kubernetes. On the infrastructure level, the chosen project is OpenStack but from the point of view of Kubernetes, the underlying infrastructure should not matter, thus this could be substituted with other projects as well.

1.8 Contribution

The most important contribution of this thesis to the IT research field is its approach to the vendor lock-in problem. It offers a tool in its early development stage to handle the problem and proves that it has the necessary capabilities to do so. The thesis might also provide its developers with valuable information for future development decisions.

1.9 Outline

The thesis is organized as follows. Chapter 2 describes the background study and relevant information needed to understand the work. The main concepts detailed here are cloud computing, containers, cloud operating systems and container orchestration tools. Chapter 3 provides the details of the methodology of the experiments. In chapter 4, the experiments and their results are presented. Chapter 5 contains a discussion about the work done, compares the results with other technologies and presents the future work. Chapter 6 concludes the thesis.

2 Background

This chapter provides the relevant background information in enough detail to clearly understand this work. It discusses different technologies that emerged in the last decade and are currently used in edge cloud systems.

2.1 Technologies

In this section, the relevant technologies of the field are described. It is important to know these technologies and understand some of the design decisions behind them.

2.1.1 Cloud Computing

In a cloud environment, in general, there are three architectural layers we need to consider:

- Software as a Service (SaaS): the user gets access to a software running in the cloud and ready to be used. In general, these software are available to multiple users and managed by the vendor.
- Platform as a Service (PaaS): the user gets the infrastructure but needs to take care of the deployment of the intended application along with all its dependencies. In this case, the user can develop and deploy applications using the provided development tools. The management of these services falls on them as well.
- Infrastructure as a Service (IaaS): the user gets virtual machines and needs to set them up for use by installing an arbitrary cloud operating system on them. From here, the users set up everything to use and need manages them themselves. The vendor in this case does little to no management work apart from ensuring that the virtual machines are up and running without interruptions.

Figure 1 shows the cloud stack and the differences between IaaS, PaaS and SaaS in terms of the layers needed to be managed by the user and the cloud vendor.

Clouds can have three main types: public, private and hybrid clouds. In the case of public clouds, a company offers different services for users. It is called public because anyone can request resources and use them. The resources can be easily scaled up or down according to the needs of the user or the application [13]. This feature is particularly advantageous due to the fact that the most common pricing strategy is to pay for the used resources only, the so called pay-per-use model [14]. The disadvantage of public clouds is privacy related. Users or organizations are unable to exactly specify where their data is stored [15]. Its security relies on the cloud provider.

Private clouds, in contrast, are accessible by one user or organization only. The point of this type of cloud is to restrict the resources and data exclusively for themselves. This provides better security and privacy. Private clouds can be achieved in two ways. One is hosted by another organization, a cloud provider, then it is called hosted or off-premise private clouds. The other is hosted by the organization itself, then it is called on-premise [15]. If it is on-premise, the organization needs to consider the initial capital investment, the cost of setting up, deploying, managing and maintaining the cloud. Private clouds are generally preferable when security or privacy critical workloads are run on the servers. Furthermore, more configuration options are available for this type of cloud.

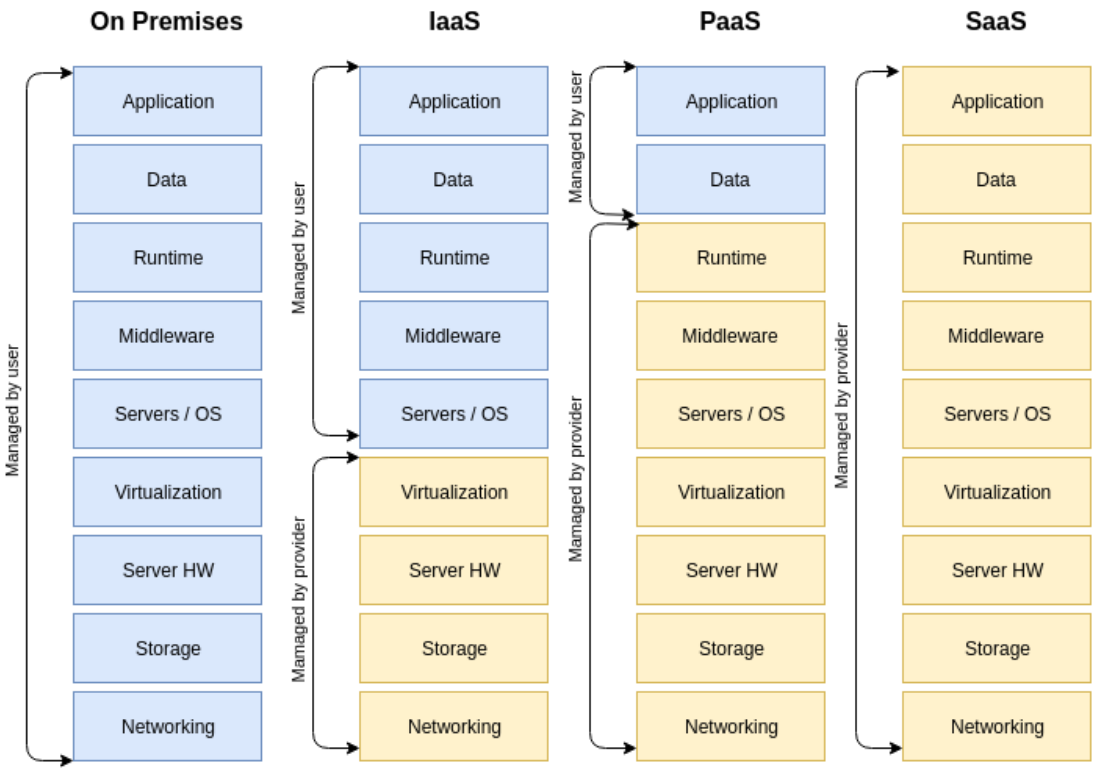


Figure 1: The cloud stack and the differences between IaaS, PaaS and SaaS. Image is based on [2][11][12]

A hybrid cloud consists of a combination of private and public clouds. This approach leverages the advantages of both types. An on-premise private cloud can store privacy critical data and run small workloads for the organization, while the public cloud can be utilized for processing computational heavy workloads [14][15].

2.1.2 Containerization

Virtual Machine (VM) technology has been a widely used virtualization technique in the past. It increases the interoperability of software and gives additional tools in the hands of its user. The virtualization of machines offers great benefits such as higher security. An application running in a VM is separated from the host machine and since the guest Operating System (OS) can be considered as untrusted, the risk of running malicious code on the host is significantly lower. Furthermore, it is capable of running software designed for an operating system different than the host.

Virtual machines, generally speaking, simulate computers. They virtualize hardware and they have their own kernels. This provides isolation from the host system. However, VMs have an unnecessarily high overhead because of this. With the rise of cloud computing and large scale systems, a better-performing, more scalable solution was needed. Thus came the concept of containers.

Containers, instead of full virtualization, use the kernel of the host machine and isolate the application and its dependencies from the host system [16]. This technology avoids the overhead of hardware virtualization as well as the need to have its own kernel. Figure 2 shows a comparison between virtual machine and container-based virtualization architecture.

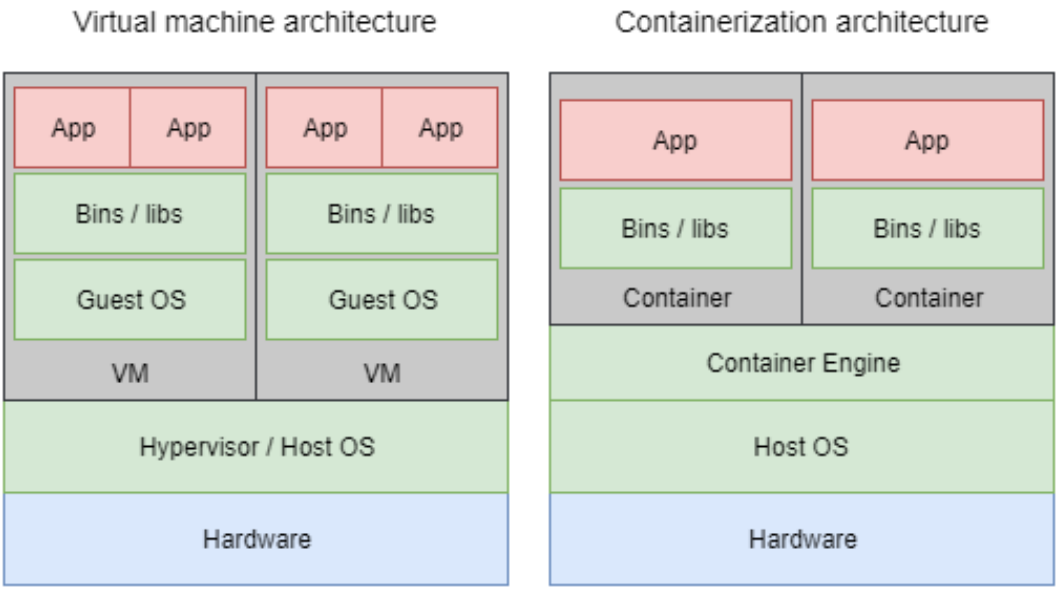


Figure 2: Virtual machine and containerization architectures [16]

According to the performance study in [17], virtual machines can take up to 1-10 minutes to start [17] compared to just a few seconds in the case of containers [18]. Comparing processing speed, containers tend to perform better, however, the extent of this can vary depending on the use case [19][20][21]. Scalability in a cloud environment is just as important as pure performance. Scaling out needs to be considered when the load for the current instances is high. In such a case, increasing the number of instances is required to handle the extra incoming traffic. Looking at this aspect, containers perform better as well. Virtual machines can take up to several minutes to scale up while containers only needed a couple of seconds. In this case, a 22-fold increase in speed could be observed [19]. However, containers still need a host OS to run on. If such a machine has no available resources to run a new container, starting up another host machine could take up significantly more time than a single container.

We can see, that containers tend to perform better than virtual machines. In a cloud environment, using containers to run applications proves to be an efficient way of virtualization. Furthermore, avoiding incompatibility is easy since the containers are isolated and contain the dependencies the app inside it needs. For instance, running two applications on a virtual machine which requires two different versions

of a software to be installed can be tricky, while simply creating two containers solves the issue. This problem is often referred to as the "dependency hell problem" [22].

2.1.3 Container Technologies

This section discusses different containerization approaches. These are:

- Linux Containers (LXC)
- Docker containers
- Kata Containers

Linux containers are part of the Linux kernel and provide operating-system-level virtualization. More specifically, the kernel supports virtualization in the userspace. It is licensed under the GNU LGPLv2.1+ [23] meaning that it is a free software. The motivation behind LXC is to provide a virtualization way similarly powerful to virtual machines but with the speed similar to operating systems running on bare metal [24]. It can be thought of as a technology in between virtual machines and chroot. Linux containers consist of multiple components which are provided by the Linux kernel [24]. The most important ones are listed here:

- Chroot - a Unix system command which provides a running application with a path that is considered as the root folder by the application. The process will be denied to access anything other than this apparent root tree [25].
- Control groups - a feature that provides a way to manage and limit resource usage for processes [26] [27].
- Namespace - Similar to control groups but it handles the resource isolation for processes. Therefore, limiting the applications to see only the allowed resources [26].

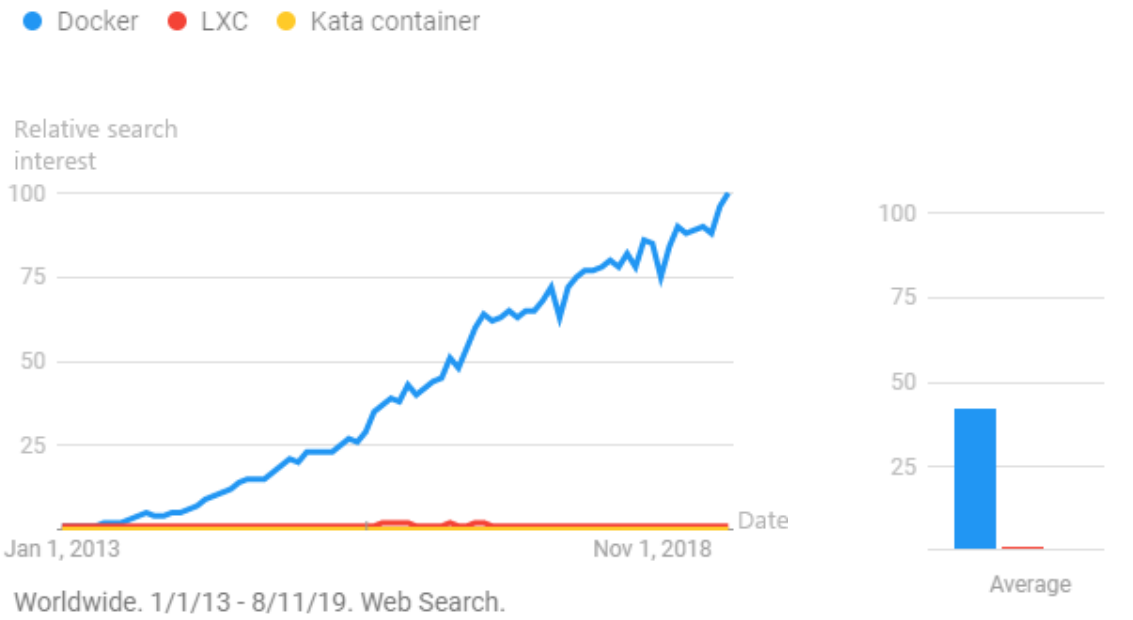
After the creation of a container, the user can access it and use it as a virtual machine, install software and run applications [28].

Docker [29] containers were introduced as an extension of Linux containers. Later, they dropped Linux containers as the default driver but kept supporting it. The new default is libcontainer which is their own project written in Go language [30]. It was made open source as well. The main component of Docker is Docker engine. This is the part which creates containers and manages the running ones. For the creation, it uses a Docker image file. This idea behind such an image is similar to a virtual machine image. It contains the dependencies and the necessary configurations. A docker image is already significantly smaller in size than a VM image, Dockerfiles can improve it even more. These files contain the dependency description for the images, environmental variables to be used and every configuration needed to run a container successfully as expected. Dockerfiles, since they are typically small text files, can be distributed easily. Furthermore, this makes it clear what the image contains and what

the container will run helping to avoid malicious software. Once we have a Dockerfile, we can build a Docker image from it and then create a container using this image.

In general, virtual machines provide higher isolation than containers, thus they tend to be more secure. Kata containers aim to leverage the security benefits of VMs over containers while keeping the performance capabilities of containers. They are more similar to a virtual machine than Docker or Linux containers. In particular, Kata containers use features of virtual machines and containers together, therefore increasing isolation and security [31]. This container runtime is open-source as well. It was launched in 2017 and still needs to mature but looks like a promising alternative to Linux kernel-based solutions for security-critical workloads.

To conclude the comparison, Docker containers seem to be the most mature and most popular technology as of now. It is open source, high performant and the default technology used in many cases. As Figure 3 shows, Docker is by far the most searched container technology on Google. Furthermore, Docker is not just a container runtime, it provides the user with multiple tools to increase the simplicity of their product.



(a) Search interest of container technologies. X axis represents the date while the Y axis represents search interest relative to the highest point on the chart. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term.

(b) Average search interest over this time period

Figure 3: Google searches for different container technologies in the last five years [32]

2.1.4 Infrastructure as a Service

In this type of service, the user can request virtual machines from the vendor. Generally, the vendor provides multiple options to choose from. These can be tiny servers with a single-core CPU and 512 MiB memory capable of running a simple web server or similar up to hundreds of cores with over ten thousand GiB of memory [33] [34]. Once the users get the machines requested, they can start setting up the upper layers. This includes tasks such as installing packages, runtimes, developer tools and dependencies, getting data to work with to the servers and developing, deploying and running applications.

In the following part of this section, the reader can find a description of different cloud operating system used in the Infrastructure as a Service bit of the cloud stack. These can be deployed to multiple hosts to provide us services such as creating virtual machines, virtual networking or usage measurement. The following projects are considered:

- Eucalyptus
- OpenNebula
- CloudStack
- OpenStack

Eucalyptus is an open-source cloud platform for building private and hybrid clouds. One benefit of this project is the compatibility with Amazon Web Services (AWS). This means that the API calls implemented in AWS can be used in the Eucalyptus on premise cloud as well. Therefore, scripts can be moved as they are too. However, this compatibility is only true for AWS. Other vendors, such as Google Cloud, IBM Cloud or Microsoft Azure still need extra work to overcome the incompatibility problems. Furthermore, Eucalyptus has a paid, commercial version which contains more features than the free one. Eucalyptus supports three hypervisors only [35]. The disadvantage of this project is its relatively limited capability to scale compared to other projects [36].

OpenNebula is a free-to-use, open-source cloud computing platform. It can be used for building private, hybrid or public cloud infrastructure. One disadvantage of this approach is the limited support for hypervisors [37] compared to other solutions. According to existing research, OpenNebula appears to be less secure, generally slower, less documented compared and not well scalable [38] [36] relative to the listed solutions. OpenNebula also requires more storage resources than the other projects and has the potential to create bottlenecks. Furthermore, their official website compares their project to OpenStack and concludes that OpenNebula is simpler to use but OpenStack provides more features [39].

CloudStack is an open-source cloud platform that can be used for public, private or hybrid cloud solutions. CloudStack supports most of the currently used hypervisors [40]. It is highly scalable and can manage high numbers of virtual machines [37]. According to previous research, CloudStack is relatively easier to deploy than OpenStack but also less stable [41]. Another disadvantage is performance. Compared to OpenStack, the difference is considerable [41].

OpenStack is an open-source, free-to-use cloud computing platform for creating private, public or hybrid clouds. One advantage of OpenStack is that it is supported by the most Linux distributions in comparison to the other projects [38] and regarding compatibility aspects, this is an important fact to consider. OpenStack is by far the most popular technology out of the ones listed here. Since the goal is to avoid vendor lock-in and switching in between technologies is difficult, it is important to target a high proportion of people. OpenStack consists of many services of which any can be used by calls to their respective Application Programming Interface (API). Having the services separated in this manner enables developers to modify or completely reimplement any service if required. This is also the reason that complicates the deployment process of OpenStack [41]. All services need to be configured separately. However, this offers greater flexibility and gives more tools in the hands of the developer. OpenStack supports the most hypervisors from this list of IaaS platforms.

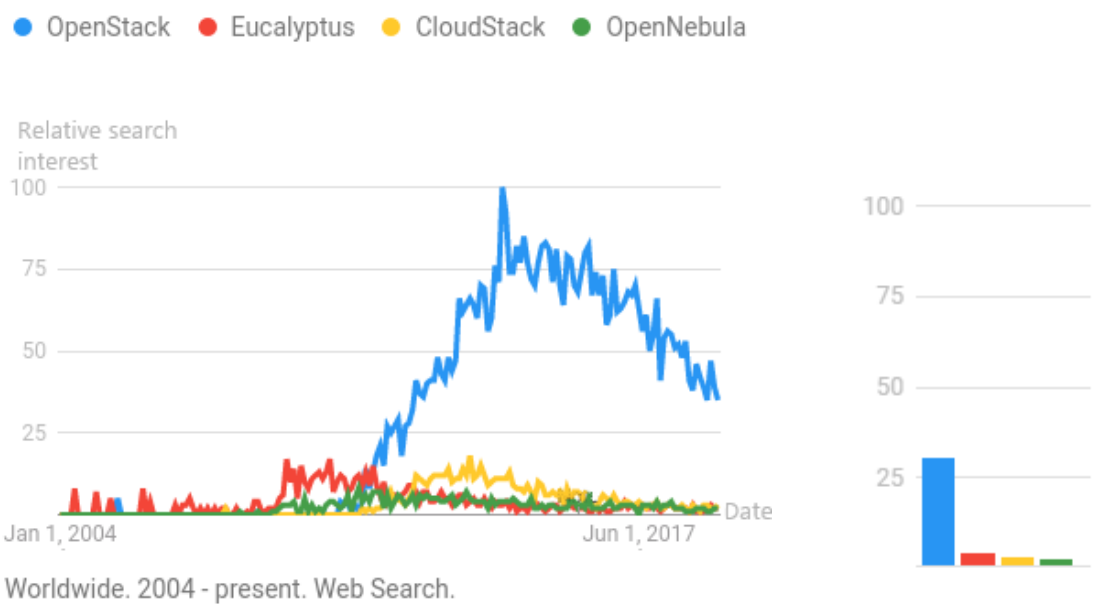
In conclusion, considering all aspects of the comparison, OpenStack presents the best results. It is the best performing, most compatible and mostly supported project out of the ones listed above and as shown in Figure 4, it is the most popular as well.

2.1.5 Platform as a Service

In this section, the Platform as a Service layer is discussed in more details. As already explained in Section 2.1.1, IaaS provides the user with a running infrastructure while handling dependencies, development, deployment and managing applications is the job of the user.

Typically, applications are run in containers here. The advantages of this approach are explained in Section 2.1.2. The most popular application container platform is Docker. With Docker, one can create Docker containers. This platform provides the user with a Graphical User Interface (GUI) and a Command Line Interface (CLI). Once the user creates an image or chooses one from a container registry, a container can be created. After successful creation, the service starts running. The user can decide to interact with the running container using either one of the provided user interfaces. Executing commands or logging into the container gives powerful tools in the hands of the user to work with Docker. This is suitable for very low-scale applications where not much management is required. However, as soon as medium or large scale operations are performed, the use of an automation tool is essential. Such tools are called container orchestration software. Some important features, among others, of an orchestration tool are:

- Creating a required number of containers



(a) Search interest in cloud computing platforms. X axis represents the date while the Y axis represents search interest relative to the highest point on the chart. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term.

(b) Average search interest over this time period

Figure 4: Graphical illustration of the popularity of cloud computing platforms measured by Google Trends based on the number of Google searches [42]

- Scaling up or down when the traffic increases or decreases, respectively
- Providing rolling updates to avoid downtime at service updates
- Handling disaster recovery

For the above reasons, orchestration tools are considered essential in some use-cases. In the rest of this section, some of these technologies will be discussed and compared in more details. The discussed projects are:

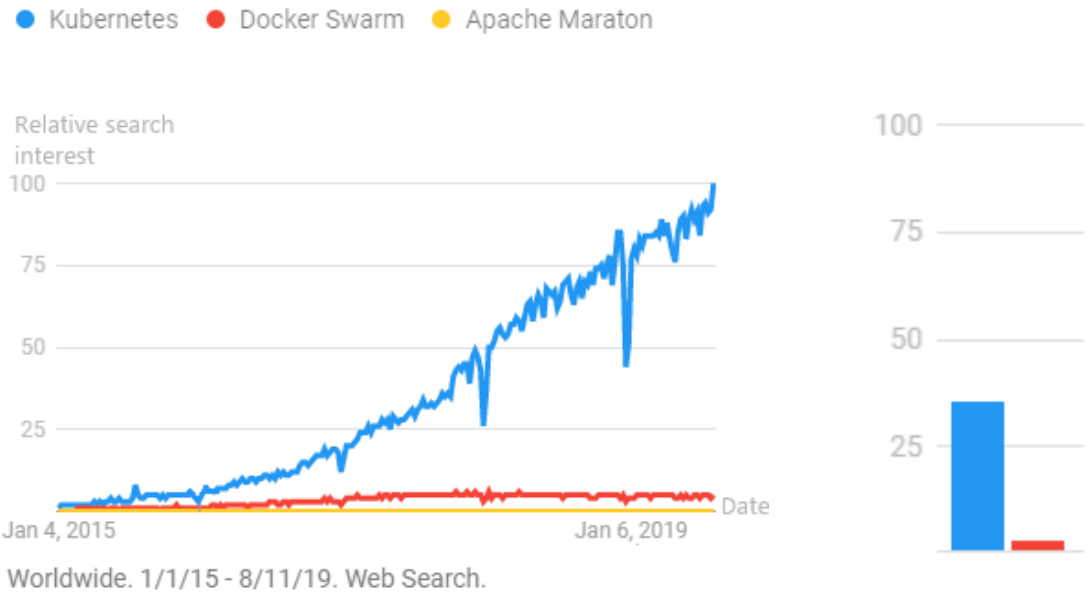
- Docker Swarm
- Apache Marathon
- Kubernetes

Docker Swarm is shipped together with Docker. It is a free-to-use and open-source software. It works by joining together a set of machines which is called a swarm. The machines in the swarm are called nodes. A node can be a manager or a worker node. Docker commands can be run on the manager node can be configured to apply them on an arbitrary set of nodes [43]. The advantage of this solution is that it is completely integrated into Docker, thus they work well together and that it is very

lightweight compared to other projects. However, it only supports Docker containers and lacks the ability to scale well [44].

Apache Marathon is an open-source container orchestration software. It supports Apache Mesos containers as well as Docker containers. Apache has multiple projects related to big data as it is an important application area for them [45]. Apache Mesos provides big data applications such as Kafka, Spark and Hadoop on each machine it runs on [46]. For this reason, Apache Marathon was designed to be highly scalable and focuses around big data. It has a two-level scheduling mechanism which is the reason behind the good scalability to even tens of thousands of nodes [47]. Furthermore, the users can interact with it through REST APIs resulting in good scriptability.

Kubernetes is an open-source container orchestration tool which is free to use. Currently, this is the most popular among such tools as shown in Figure 5.



(a) Search interest of container orchestration tools. X axis represents the date while the Y axis represents search interest relative to the highest point on the chart. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term.

(b) Average search interest over this time period

Figure 5: Graphical illustration of the popularity of container orchestration tools measured by Google Trends based on the number of Google searches [48]

A feature called Container Runtime Interface (CRI) was added to Kubernetes which is responsible for communicating with the actual container runtime. The benefit of such an approach is that support for new runtimes can be added relatively simply to the CRI without the need to recompile the entire Kubernetes [49]. Figure 6 shows a simple representation of how CRI fits into the Kubernetes system. As we can see, it communicates with the Kubelet agent and the container runtime in use. It

works in between them and translates the instructions from Kubelet to be understood by the appropriate container runtime. The Kubelet module of Kubernetes is the main node agent running on every node. It is responsible for creating and running pods. In Kubernetes, pods are the minimal deployable entities which consist of one or more containers.

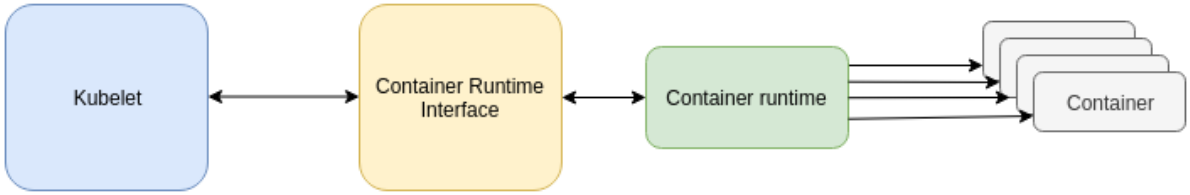


Figure 6: Illustration of the Kubernetes CRI relative to other components of Kubernetes. Image based on [49]

A Kubernetes cluster consists of a master node and worker nodes. The master manages the applications running on the worker nodes [50]. Kubernetes is well scalable and supports large clusters as well. According to their website [51], now they support a cluster if it contains no more than:

- 5000 nodes
- 150,000 pods in total
- 300,000 containers in total
- 100 pods on a single node

Disadvantage of Kubernetes is its complexity. It contains a large number of components and configuration options.

In conclusion, Kubernetes is well-scalable, currently the most popular IaaS project, robust, mature and it has support to many other technologies. It fits the best for the purposes of this thesis.

2.1.5.1 Federation Kubernetes cluster federation is a tool aiming to facilitate the management of multiple clusters. It achieves this by creating a cluster federation where one cluster is appointed to be the host cluster. Others can be joined to this federation which makes them member clusters. The main features of federation are [52]:

- Synchronizing and automatically managing resources among multiple clusters. The user can decide to propagate a resource to all of the clusters or just some of them and the federation will take care of it and maintain the required number of replicas without the need to manually configure it in all clusters.

- Providing a Domain Name Service (DNS) that can discover services across clusters. An example case where this can be useful is when the frontend of an application runs in every cluster in order to be as fast-responding as possible for the user but a database is stored in a central data center that occasionally needs to be accessed.

In the history of Federation, we can distinguish two projects: Federation v1 and v2. The earlier version was designed to use the Kubernetes API as it is. It seemed to provide sufficient functionality for the purposes of the project but this turned out to be false. The project was abandoned and the Federation v2 project was launched which aims to be able to solve the more complex problems Federation v1 could not tackle. Federation v2 is in prototype phase [53], as of writing this thesis no beta version has been released yet but it is planned to happen before the end of 2019. In this thesis, the focus is on Federation v2 because researching its capabilities in this early stage can benefit the users as well as the developers of the project.

Kubernetes Federation v2 requires configuration when a user intends to set up a federation control plane. It relies on two main kinds of information that needs to be configured, namely cluster configuration and type configuration, also shown in Figure 7. Cluster configuration instructs the federation control plane about the clusters it needs to target, whereas type configuration states what types of Kubernetes APIs it should handle. Type configuration has three building blocks which need to be considered:

- Template: declaration and configuration of a resource that the federation will create in the clusters.
- Placement: a configuration that tells the federation which are the target clusters for a specific resource.
- Override: cluster-specific configuration for fine tuning the template

Once the federation is configured with this information, it can start the propagation of resources. This means that the host cluster distributes them among the member clusters. Furthermore, the mentioned building blocks consist of three fundamental parts, namely status, policy and scheduling. Status refers to the status of the propagated resources, policy defines the permitted target clusters for these resources, whereas, taking these into account, scheduling decides about the distribution of workloads.

3 Methods

This chapter describes the details of the environment, the experiment in general, and how the cloud stack is built up.

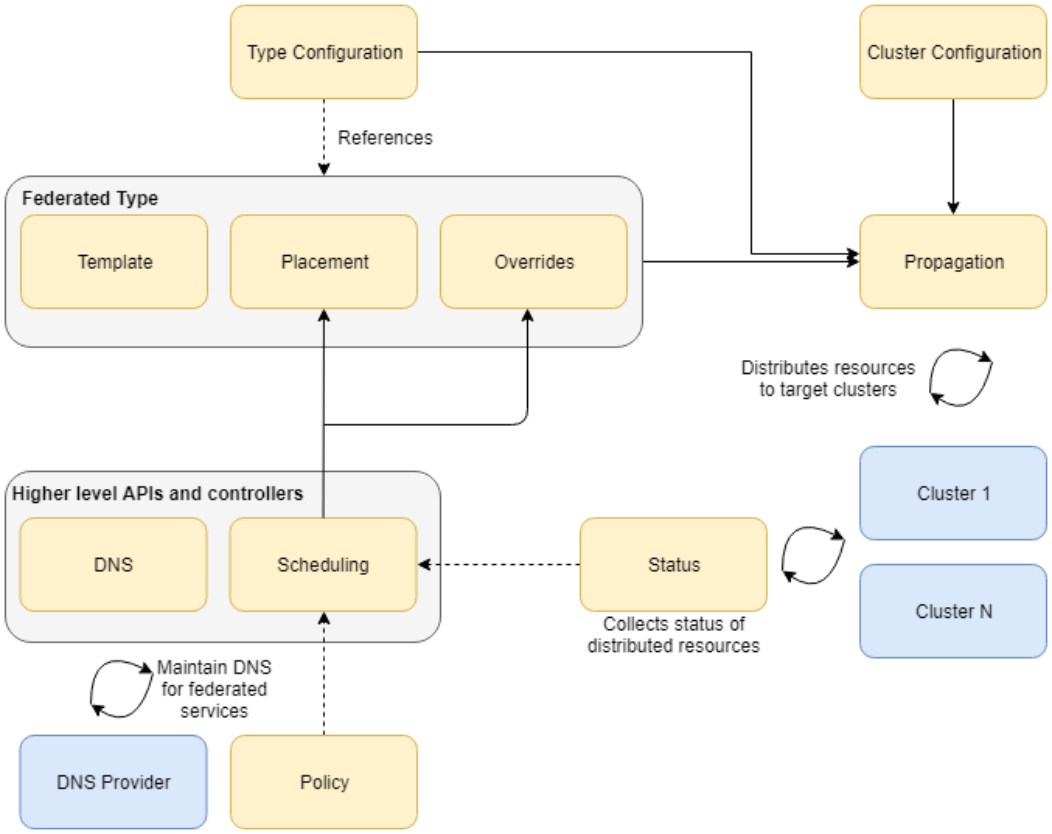


Figure 7: Kubernetes Federation v2 concepts [53]

3.1 Cloud environment

The resources needed to carry out the project is provided by CityNetwork[54]. Their infrastructure layer is deployed using OpenStack. A project on this infrastructure, also called a tenant, was created for the purposes of the thesis. The total available resources for this tenant are:

- 100 VCPUs
- 200 GB RAM
- 1000 GB storage

Said infrastructure is used for the thesis. For the experiments, it is considered to be bare metal with the difference that all networking tools are virtualized. That said, virtual routers, networks and instances need to be created. Once the instances are up and running and communication works in between them, our own OpenStack infrastructure can be deployed and considered as an edge. Continuing the stack, Kubernetes is installed on top of it and it is used for running the experimental applications.

OpenStack is deployed using Kolla-Ansible[55]. Kolla-Ansible is a relatively easy-to-use tool for deploying OpenStack in Docker containers. Kolla-Ansible has a network requirement of two interfaces: one for providing internet for the instances and one for the OpenStack Neutron network. Neutron is a component of OpenStack responsible for networking. One example network that suits the needs of Kolla-Ansible is shown in Figure 8. In the experimental setup, the different edges might have different number of instances but they all follow the same idea to have:

- One jumphost
- One controller node
- Multiple compute node

In this setup, a jumphost is an instance with a floating IP address which is public. It can be reached from anywhere. The other hosts have no public IP addresses and can only be reached from within the same network they are connected to. The jumphost can reach the controller and compute nodes, thus it is used to manage the system. The user is authenticated on the jumphost with SSH keys. Then, from the jumphost to the other nodes, SSH key-based authentication is used again. Password authentication is disabled for security reasons. In OpenStack, there are two basic types of nodes. The main task of a compute node is to run a hypervisor that can spin up and run virtual machines when it is told so. Other tasks are carried out by the controller node, such as scheduling, networking, running database servers or APIs [56]. It is possible to have multiple controller nodes that can be dedicated to certain tasks. For example, a network controller node. In our setup, there is one controller node that performs all the necessary work to keep the system simple.

3.2 Platform layer

For the platform layer, Kubernetes is deployed on OpenStack. There are multiple ways to execute the deployment but similarly to OpenStack, unless advanced configurations are required, any tool is capable of performing this task. For this thesis, a tool called Kubespray [58] is used. Kubespray is a Kubernetes side-project [59] which is a simple, easily configurable and flexible tool for deploying Kubernetes clusters. It uses Ansible for the process and it runs Kubernetes services in Docker containers on the machines it is deployed on.

The setup is very similar to the one OpenStack is deployed on. At least three virtual machines are created. One as a jumphost that will communicate with the other ones, initiate the deployment and manage the cluster. One manager node, called a Kubernetes master node, which similarly to an OpenStack controller node, handles API calls, runs interfaces and schedules workloads. Furthermore, one or more Kubernetes worker nodes are added as well, that creates, runs and manages containerized applications as Docker containers.

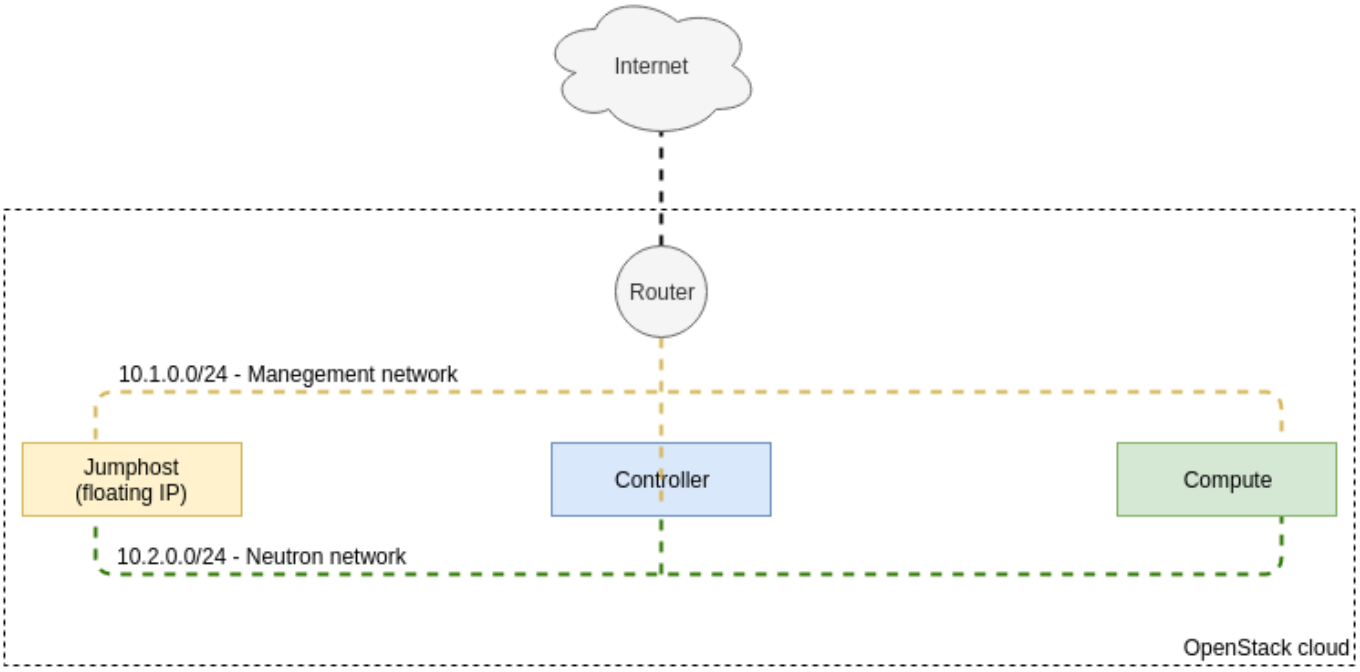


Figure 8: OpenStack virtual network example based on the requirements in the documentation [57]

Once we have decided on a setup and created a necessary virtual machines that Kubernetes will run on, we need to tell Kubernetes this information. This information is fed to Kubernetes through Ansible inventory files [60]. An inventory is a configuration file used for telling Ansible about the hosts it is supposed to work on. Listing 1 shows how a simple, example inventory file looks like in case of one master and one worker node. In such a file, we need to list all the nodes under *[all]* that are going to be part of the cluster. *[kube-master]* is a list of the master nodes and *[kube-node]* is a list of worker nodes. Furthermore, we need to specify which node will run the etcd container [61], which is a distributed key-value storage used for storing Kubernetes cluster-state and all necessary information required for its operation.

```
[k8s-cluster:children]
kube-master
kube-node

[all]
controller ansible_user=ubuntu ansible_become=true
compute ansible_user=ubuntu ansible_become=true

[kube-master]
```

```
controller

[kube-node]
controller
compute

[etcd]
controller
```

Listing 1: Sample inventory file for deploying Kubernetes with Kubespray

One cluster is now defined, but for the experiments another one is required as well. To test the major point of the thesis, the avoidance of vendor lock-in, the other cluster is hosted on another public cloud provider. This other cloud provider is Microsoft Azure [62]. The Azure Dashboard provides a graphical interface to create a Kubernetes cluster with just a few clicks. The cluster created consists of five nodes, all running Kubernetes v1.12.7. Note that this version is different from the other cluster to test whether the assumption that different versions cause no problems is true or not. The name of the size of each node is *Standard DS2 v2* denoting a machine with two VCPUs and seven GB of memory. The name and the context of this cluster are *aks-cluster*.

Commands are executed using the kubectl tool. Kubectl is a CLI for running commands against Kubernetes clusters[63]. It is used, among other things, for creating, running, managing applications, configuring deployed clusters or gathering information about them. It needs to be configured to know about the cluster it needs to operate against. A so called kubeconfig file serves this purpose. Listing 2 shows an example kubeconfig file. It needs to be in YAML language. This file contains the address of the API server as well as the name of the cluster and its context. Furthermore, it knows the key and certificate needed for secure communication with the API server.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://10.1.0.40:6443
  name: gabor.host
contexts:
- context:
    cluster: gabor.cluster
    user: admin-gabor.cluster
  name: cluster1
current-context: cluster1
kind: Config
preferences: {}
users:
```

```
- name: admin-gabor.cluster
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

Listing 2: Sample kubeconfig file for configuring kubectl command line tool

If multiple clusters can be reached from a single host, multiple kubeconfigs can be provided to kubectl. In this case, a context can be specified in order to execute a command on the right cluster.

3.3 Federation

Kubernetes Federation v2 requires at least two clusters to have benefits. The two clusters are the ones defined in Section 3.2. Both of these two clusters can be accessed by the jumphost and the commands are executed from this host as well. One of the clusters needs to be appointed as host cluster, while the other one joining the federation is a member cluster. Kubernetes Federation v2 is supposed to work with any cluster with version above v1.11. Conducting the experiments with different versions of Kubernetes adds important insights to the results because expecting all clusters on every edge running on different cloud providers' systems to have the same version is unrealistic. The promising characteristic of Federation v2 is its little requirements of clusters. In between edges it is expected to have differences. Furthermore, member clusters of a federation can be managed by a single host without the need to manually configure them one by one. This is especially beneficial in edge environments where a high number of clusters can be expected.

3.4 Approach of the experiments

The experiments intend to mitigate the vendor lock-in problem on the platform layer. The aim of the thesis is to discover and propose a way to utilize a combination of tools and prove its effectiveness in this area. In order to be able to prove this, a certain set of application needs to be defined and the experiments need to be carried out on a general item of this set.

This thesis considers a classification of applications which divides them into two classes, namely stateful and stateless. Stateless application requires no information from earlier sessions. It relies entirely on its source code and the data fed to it in a single session. In contrast, stateful applications rely on data from earlier sessions [64]. It can be, for instance, a database containing entries from users, such as a forum, or metadata identifying the users to keep them logged in. This class of applications requires a data storage that the replicas can access. This can result in a bottleneck of performance due to either geographical distance or the requirement of transactional atomicity [64]. Distributed databases, such as Redis, aim to tackle this problem by running a master and multiple slave servers [65]. From an application migration perspective, this introduces additional experimental questions, such as how to move

the data together with the application. Due to these facts, stateless applications tend to scale better, whereas stateful ones tend to perform better in security aspects [66].

Considering the classification discussed here, this thesis focuses on the stateless application class. From an application migration perspective, the successful outcome of this class is a prerequisite for the stateful one. Therefore, experiments targeting the stateless class are performed.

For the experiments, an application is designed with stateless characteristics. It is designed with the idea to keep it simple yet demonstrate the use case of stateless applications that a user provides it with data which gets processed and the results are sent back to the user. Two experiments are performed. The first one is a single, standalone application and the second one is a combination of applications where they need to communicate with each other. The first experiment proves the basis of the idea while the second one tests whether a set of applications can work together in a stateless manner even after being migrated to a different cluster.

The experimental application waits for an input number, finds the closest prime number that is equal to or lower than the input and returns that value. To be able to run such an application on our federated clusters, there are multiple steps to take including:

1. Writing the code of the application
2. Creating a Docker image out of it
3. Writing Kubernetes deployment and service YAML files first to make sure it runs in a single cluster
4. Federating the resources

The second application uses the first one as a service. The user gives an input to the second application which transfers it to the first one. Then the first application returns the closest smaller prime number. The second application takes this number as an input variable n and returns the n^{th} prime number. The steps to create this application are similar to the first one.

4 Experiments and results

This chapter describes how the experimental environment for this thesis is created, how experiments are carried out, and the outcomes.

4.1 Experimental setup

The resources discussed in this section are all virtualized and created in the cloud infrastructure. A router called *gabor-router* is connected to the internet and another

network, called *gabor-net1*. This network will provide the internet for the instances created on it. A subnet with 10.1.0.0/24 prefix specifies the IPs available for use. Another network called *gabor-net2* is used by the OpenStack Neutron component for its trafficking. The subnet, similarly to the one mentioned above, has the prefix 10.2.0.0/24.

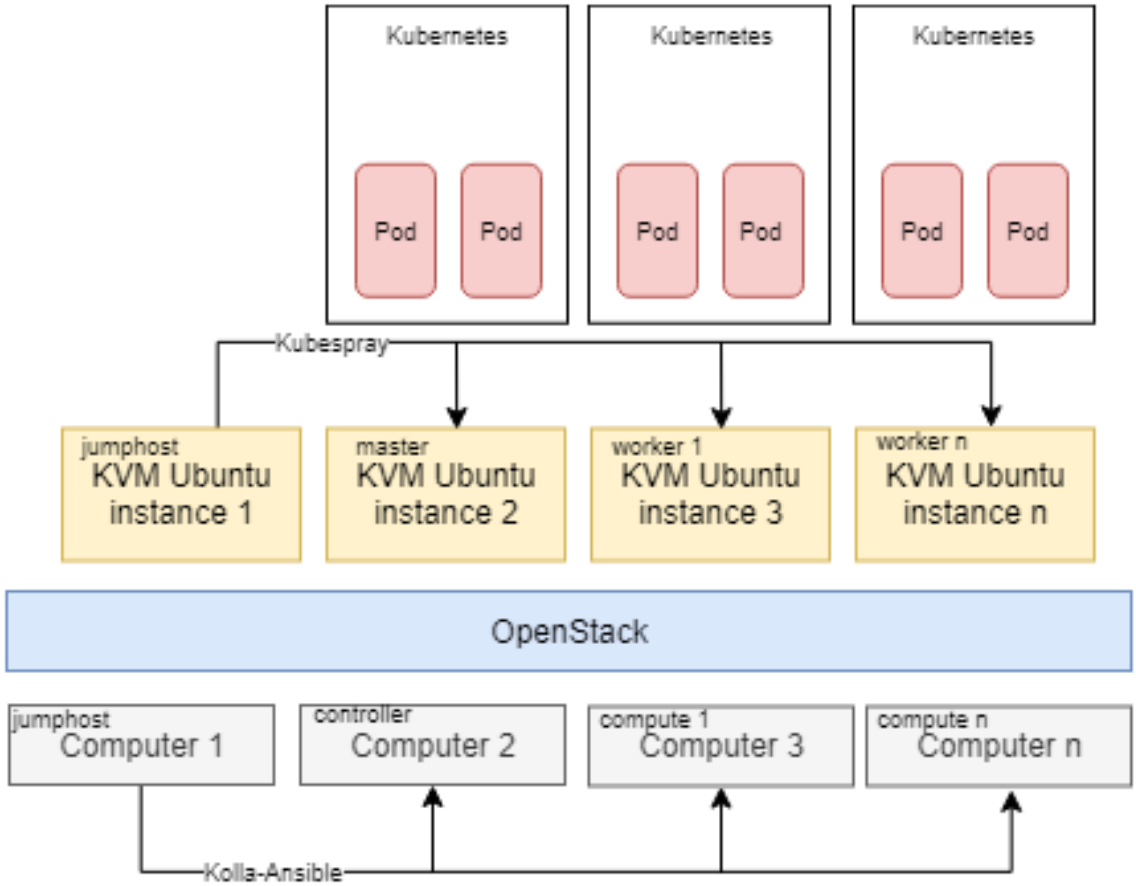


Figure 9: Graphical illustration of a cluster in the experimental setup used in the thesis

Figure 9 shows the setup described in this section. In the first steps, OpenStack needs to be deployed. The cloud providers made some computers available for the purpose of this thesis. One of these is the jumphost, called gabor-jumphost which is the main host used in the setup. An important purpose of this is to avoid logging into every host and instead run automated scripts from this computer. This is essential in a cloud environment since the number of computers can be very high, thus a user simply cannot handle them all one-by-one. Four other instances are created connected to the *gabor-net1* and *gabor-net2*. Namely, these are *gabor-controller2*, *gabor-compute2*, *gabor-compute3* and *gabor-compute4*. Their names already suggest their purpose. *gabor-controller2* is used as an OpenStack controller while the other instances are compute nodes. These instances are created with an image of Ubuntu

16.04. The reason for this particular operating system is that its highly compatible with most tools. Little extra work is needed to deploy OpenStack or Kubernetes on it since it is supported by these projects as well as deployment tools. Resources allocated to each instance are four CPUs, 8 GB RAM and 40 GB disk space. This is sufficient for our experiments. Using kolla-ansible from the jumphost, OpenStack is deployed on these four hosts. Since kolla-ansible uses Ansible as a deployment tool, it needs to be provided an inventory file. This inventory file is long but the most important lines specifying the hosts are shown in Listing 3.

```
[control]
gabor-controller2 ansible_user=ubuntu
    ansible_become=true
[compute]
gabor-compute2 ansible_user=ubuntu
    ansible_become=true
gabor-compute3 ansible_user=ubuntu
    ansible_become=true
gabor-compute4 ansible_user=ubuntu
    ansible_become=true
```

Listing 3: Part of the inventory file used for deploying OpenStack with kolla-ansible

After deployment, we can see docker containers running on both the controller and compute nodes. Listing 4 shows some of the running containers on the controller node.

```
92b1894102ec kolla/ubuntu-source-horizon:rocky
    "dumb-init --single-c" horizon
cbc200157d6b kolla/ubuntu-source-neutron-server:rocky
    "dumb-init --single-c" neutron_server
903ddbe94fa0 kolla/ubuntu-source-nova-api:rocky
    "dumb-init --single-c" nova_api
4ec6dacdb4c7 kolla/ubuntu-source-fluentd:rocky
    "dumb-init --single-c" fluentd
```

Listing 4: A sample of OpenStack services running in Docker containers on the controller node.

With OpenStack deployed, the next step is Kubernetes. This process is also shown in Figure 9. On our OpenStack we create a similar setup to the one discussed above with one router and two networks. However, only three instances are created for Kubernetes. One jumphost, one Kubernetes master and one Kubernetes worker node. They are called jumphost, controller and compute, respectively. With these machines up and running, Kubernetes can be deployed on them. Kubespray is used for this process. Since Kubespray uses Ansible, an inventory file needs to be provided similarly to the case of kolla-ansible. The experimental cluster is very simple, an inventory file similar to the one shown by Listing 1 can be passed to Kubespray. Kubespray can be configured to install kubectl on the machine the script is run on, thus no manual installation is needed. After the script is executed, the kubeconfig

file can be found on the controller node or, if Kubespray was configured so, then it is automatically copied to the deployment host. Logging into the master node and listing the running containers, it shows Kubernetes services running in Docker containers. A sample of such a list is shown by Listing 5.

```
9584265cb1fd "/dashboard --inse..."
    k8s_kubernetes-dashboard_kubernetes-dashboard
16f1e220d136 "/hyperkube contro..."
    k8s_kube-controller-manager_kube-controller-manager
9f1b449f268b "/kube-dns --domai..."
    k8s_kubedns_kube-dns
01a6bf72d457 "/hyperkube proxy ..."
    k8s_kube-proxy_kube-proxy-controller
4e3c696cae4c "/usr/local/bin/etcd"    etcd1
```

Listing 5: A sample of Kubernetes services running in Docker containers on the master node

In order to create a cluster federation, a second cluster is deployed with similar configuration to the first one. The clusters are named according to their status in the federation: *gabor.host* and *aks-cluster*. The *gabor.host* cluster runs the federation control plane and therefore acts as the single source of truth for the members. The *aks-cluster* cluster is joined to this federation. In order to test the compatibility of different versions as well, *gabor.host* runs Kubernetes version v1.11.3 while *aks-cluster* runs v1.12.7. Listing 9 in Appendix A shows the command to create a cluster federation and its output. The first time the command is run, the host cluster is created. This is what the snippet shows.

4.2 Experimental applications

The applications are implemented in Python 3 language. The code for the first one is shown in Listing 6. Python 3 is simple and often used for backend servers for processing data. The application relies on the REST protocol. Since the REST protocol is stateless [67], it fits our purposes. The user needs to send a GET request for the application to a subdomain identical to the number the user wishes to transfer. An example with cURL [68] looks like the following:

```
$ curl -X GET 127.0.0.1:5000/10
```

In this example case, the application is running on the localhost, listening on port 5000 and the number 10 is sent to it as input. The code imports the Flask package. Flask is a microframework [69], which in this case is used for defining endpoints. An integer argument is passed to the *get_prime* function. This function then returns the closest smaller or equal prime number using the SymPy package. SymPy is a Python library containing functions for symbolic mathematics[70]. In this applications, two functions are used, namely *isprime* and *prevprime*. The function *isprime* decides if a number is prime or not whereas *prevprime* returns the closest previous prime number relative to its parameter. Eventually, when the prime number has been specified, it gets sent back to the client as a response to the GET request.

```
from flask import Flask
import sympy

app = Flask(__name__)

@app.route('/<int:number>', methods=['GET'])
def get_prime(number):
  number = int(number)
  if number < 2:
    return "No smaller prime number exists\n"
  else:
    retstr = "Closest smaller prime number: "
    retnum = number
    if not sympy.isprime(number):
      retnum = sympy.prevprime(number)
    return retstr + str(retnum) + "\n"

app.run(host='0.0.0.0', debug=False)
```

Listing 6: The experimental Python 3 application that returns the closest smaller prime to the input number

When the application itself is implemented, the next step is to create a Docker image out of it. This is performed using a Dockerfile and the *docker build* command. The contents of this file is shown in Listing 7. It specifies the base image to use, which in this case is Python 3.7. The *ADD* command copies the content of the current folder to the */code* folder in the container. This only contains the python code. The *WORKDIR* command sets the working directory in the container to the */code* path. Then, necessary packages are installed using pip. Port 5000 is exposed, therefore the container listens on this port. Important to note, that this command only exposes the port, as a kind of documentation for the programmer, but it is not published. Eventually, the startup commands for the container is specified, which simply starts up our application.

```
FROM python:3.7-alpine
ADD . /code
WORKDIR /code
RUN pip install flask sympy
EXPOSE 5000
CMD ["python", "app.py"]
```

Listing 7: The Dockerfile for building and image from the python application

This Dockerfile is built into an image using the following command:

```
$ docker build -t finta/edgethesispythonapp .
```

The created image is pushed and stored in Docker Hub[71] in the *finta/edgethesispythonapp* repository. Kubernetes will use this as the image name to download

the image at application startup.

The next step is to create a YAML file that tells Kubernetes what resources are required to be used and how to configure them. Listing 8 shows the contents of this file. Two resources are defined in it: deployment and service. In general, a deployment definition contains the specification of a Kubernetes pod and takes care of fulfilling the requirements defined in it. Here, the specification says that the pod needs to have a container in it which uses the image with the *finta/edgethesispythonapp:latest* tag. Furthermore, this container exposes the port 5000. The number of replicas defines how many copies of this pod need to be created. Increasing this number increases the available backend pods running, thus resulting in less workload for each of them. In such a case, Kubernetes itself takes care of load-balancing the workload among the pods. A service is also defined. A Kubernetes service serves the purpose of easily reaching pods. Pods have their own IP addresses which can be used to reach them but new pods can be created with new IPs or old ones can be removed anytime. Therefore, an extra layer of abstraction is needed to reliably access the pods. When the user communicates with a service, it picks one of the underlying pods and forwards the traffic to that one. The set of pods are usually linked to a service with the help of selector. In this case, the selector is the *app: python-app* key-value label. Some ports need to be defined in the specification. The *targetPort* defines which port the traffic will be sent to in the container. This is the same as the port exposed earlier. The value of the *port* key defines which port is exposed inside the Kubernetes cluster. This can be accessed by other services if needed. Finally, *nodePort* exposes a port to the outside of the cluster making it available to be accessed by a user. For this key, port 31111 is chosen as the value. If it was left unset, it would be an automatically and randomly generated number between 30000 and 32767 [72]. Setting this port to a fixed value makes it easier to access the application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: python-deployment
  namespace: python-ns
  labels:
    app: python-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: python-app
  template:
    metadata:
      labels:
        app: python-app
    spec:
```

```
      containers:
      - name: python-app
        image: finta/edgethesispythonapp:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: python-service
  namespace: python-ns
spec:
  ports:
  - nodePort: 31111
    targetPort: 5000
    port: 5000
    protocol: TCP
  selector:
    app: python-app
  type: NodePort
```

Listing 8: The YAML file for the Python 3 application that Kubernetes uses to create the necessary resources

To test if this configuration works, the resources can be created by executing the following commands against a cluster:

```
$ kubectl create ns python-ns
$ kubectl apply -f pythonprime.yaml
```

Now the resources should be up and running. Figure 10 shows how this can be checked and made sure of. By default, security rules restrict sending GET requests

```
ubuntu@gabor-jumphost:~/pythonprime$ kubectl get pods -n python-ns
NAME                                 READY     STATUS    RESTARTS   AGE
python-deployment-55d965fb96-dwdwj   1/1       Running   0          8m
ubuntu@gabor-jumphost:~/pythonprime$ kubectl get deployments -n python-ns
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
python-deployment   1         1         1            1           8m
ubuntu@gabor-jumphost:~/pythonprime$ kubectl get services -n python-ns
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
python-service   NodePort   10.233.44.196   <none>        5000:31111/TCP   8m
```

Figure 10: Graphical representation of commands showing the Python application running in a Kubernetes cluster

from one instance to another on port 31111, thus this port needs to be enabled in OpenStack. This is true to any other application where the user shall try to access an application running on another instance. In this particular case, the Kubernetes

master node is running on gabor-controller1, thus the deployment can be tested by sending a GET request to this host on the port 31111.

```
$ curl gabor-controller1:31111/20
Closest smaller prime number: 19
```

This shows that the application works fine in a single cluster.

The next step is to federate these resources. The YAML files are modified and some other ones are created as well. Four YAML files describe the basic specification of the application as shown in Appendix B. Here, the non-federated namespace is defined in a YAML file, as shown in Listing 10, to reduce the manual execution of commands, therefore making automation simpler. This namespace then needs to be federated. The corresponding configuration is shown in Listing 11. The federated deployment and service are shown in Listing 12 and Listing 13, respectively. There is a new mapping in these files called *placement*. This specifies which clusters the corresponding resources are expected to be propagated to. The placements type was discussed in more details in Section 2.1.5.1. This application is used for the experiments. In the FederatedService definition, overrides are defined as well. Microsoft Azure supports load balancers, thus this is used to automatically allocate a public IP address for the running application which can be used to access it from outside the cluster.

Migration of the application is carried out by modifying the placement. As it can be seen in Appendix B, the placement of the deployment and service resources are set to both the *gabor.host* and *aks-cluster* clusters. Therefore, resources are expected to be propagated to both of them. However, the federated namespace *python-ns*, in which these resources exist, is only propagated to the *gabor.host* cluster. No federated namespace is created in the other cluster, limiting the resources to this one only. The namespace, the deployment and the service are created using the *apply* command:

```
$ kubectl apply -f namespace.yaml -f federatednamespace.yaml \
    -f federateddeployment.yaml -f federatedservice.yaml
namespace/python-ns created
federatednamespace.types.federation.k8s.io/python-ns created
federateddeployment.types.federation.k8s.io/python-deployment
    created
federatedservice.types.federation.k8s.io/python-service created
```

The federated namespace now exists only in the *gabor.host* cluster. In this namespace, the deployment forces the images for the containers to be pulled from the repository, creates the containers encapsulated in pods and makes sure the required number of replicas always run, in this case one only. The service creates an endpoint for the pod and exposes the port 31111. We need to make sure that everything happens as expected. The available resources are listed for the clusters.

```
$ kubectl get namespace --context=gabor.host | grep python-ns
NAME          STATUS   AGE
```

```
python-ns    Active   24m

$ kubectl get deployment -n python-ns --context=gabor.host
NAME                 DESIRED   CURRENT   UP-TO-DATE
python-deployment    1         1         1
    AVAILABLE   AGE
    1           26m

$ kubectl get federatednamespace -n python-ns --context=gabor.host
NAME         AGE
python-ns    26m

$ kubectl get services -n python-ns --context=gabor.host
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP
python-service    NodePort    10.233.18.56    <none>
    PORT(S)           AGE
    5000:31111/TCP    26m

$ kubectl get pods -n python-ns --context=gabor.host
NAME                                  READY     STATUS
python-deployment-55d965fb96-9xxhg    1/1       Running
    RESTARTS   AGE
    0          28m
```

The following command returns no namespaces.

```
kubectl get namespace --context=aks-cluster | grep python-ns
```

Federation v2 successfully placed the resources to *gabor.host* while it did not propagate them to *aks-cluster* as configured. Sending a GET request to the endpoint, the result is returned as expected.

```
$ curl gabor-controller1:31111/18
Closest smaller prime number: 17
```

Migration of the application happens by propagating the resources to other clusters. The deployment and service are already configured for this, only the namespace is needed to be changed. A patch is executed on the resource that changes the placement.

```
$ kubectl -n python-ns edit federatednamespace python-ns
federatednamespace.types.federation.k8s.io/python-ns edited
```

The *edit* command opens a vi editor on the given resource. Now the target cluster is changed to *aks-cluster* manually. The federation controller-manager recognizes this change and propagates the resources to the given clusters. The resources can be queried again.

```
$ kubectl get ns --context=aks-cluster | grep python-ns
python-ns            Active     7m2s

$ kubectl get deployment -n python-ns --context=aks-cluster
```

```
NAME                    DESIRED   CURRENT   UP-TO-DATE
python-deployment       1         1         1
    AVAILABLE   AGE
    1           7m12s

$ kubectl get pod -n python-ns --context=aks-cluster
NAME                                     READY     STATUS
python-deployment-ccf59b558-5dmql        1/1       Running
    RESTARTS   AGE
    0          7m18s

$ kubectl get service -n python-ns --context=aks-cluster
NAME             TYPE           CLUSTER-IP    EXTERNAL-IP
python-service   LoadBalancer   10.0.35.55    52.236.33.215
    PORT(S)           AGE
    31111:31232/TCP   7m26s
```

The resources are up and running in *aks-cluster*. The service in this particular case took around one minute to generate an external IP address. This process can take up to several minutes. The LoadBalancer type of the service means that the overrides defined in the YAML file was successful as well. The application can be tested by sending a GET request to the external IP address.

```
$ curl 52.236.33.215:31111/28
Closest smaller prime number: 23
```

The application now runs in *aks-cluster*. We need to make sure that it has been completely migrated and no resources are running in the other cluster.

```
$ kubectl get pod -n python-ns --context=gabor.host
No resources found.

$ kubectl get deployment -n python-ns --context=gabor.host
No resources found.

$ kubectl get service -n python-ns --context=gabor.host
No resources found.
```

The application has been successfully migrated to another cluster running in another cloud provider's cloud. The application can be moved back to the other cluster with similar steps.

The second application is tested in the following part. It is important to be able to migrate multiple applications at the same time without manual steps, otherwise it can get complex in a system with multiple components. The non-federated files for the second application are listed in Appendix C. Listing 14 contains the Python code. It is similar to the first one, but in this case an extra package, *requests* is imported for making HTTP requests. The address for the request is set to *python-service* because the Kubernetes service in named this as seen in Listing 8. That service creates an endpoint only available inside the cluster and this application communicates with

the other one through this. From the response, the number is extracted and passed to the *sympy.prime*[73] function as an argument. It returns the n^{th} prime number and this is the result of the algorithm. The *Dockerfile* is very similar to the first one. The only difference is a single python package that needs to be installed. Listing 16 shows the YAML file for the application. It differs from the first application in the name, the container image, the port and the nodePort. This application is exposed on port 31112.

First, the applications are tested on a single cluster as non-federated resources. After creating a namespace, the applications can be deployed, as shown on Figure 11. Important to note that for this experiment, the first application is modified. The type of the service is set to ClusterIP to simulate a backend application which is not exposed to the internet and is reachable from inside the cluster only. The application

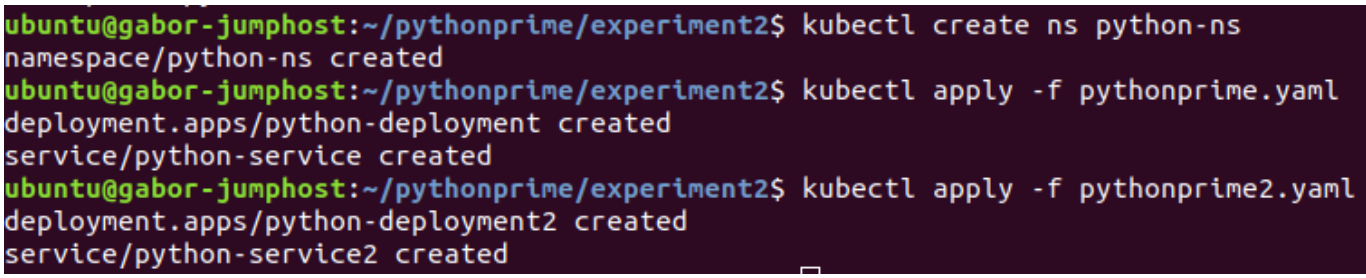


Figure 11: Deployment of the two applications

is listening on port 31112, as specified in the service definition. The application can be tested with a GET request.

```
$ curl gabor-controller1:31112/4
3. prime number: 5
```

In this case, the second application gets the input number 4, and calls the first application with this input. It gets a response with 3 as the closes smaller prime number and returns the third prime number which is 5. This shows that the application works as intended.

The files for the federated second application can be found in Appendix D. Listing 17 shows the federated deployment while Listing 18 shows the federated service resource.

The deployment command of the two applications together is executed similarly as before.

```
$ kubectl apply -f namespace.yaml -f federatednamespace.yaml \
    -f federateddeployment.yaml -f federatedservice.yaml \
    -f federateddeployment2.yaml -f federatedservice2.yaml
namespace/python-ns created
federatednamespace.types.federation.k8s.io/python-ns
    created
```

```
federateddeployment.types.federation.k8s.io/python-deployment
    created
federatedservice.types.federation.k8s.io/python-service
    created
federateddeployment.types.federation.k8s.io/python-deployment2
    created
federatedservice.types.federation.k8s.io/python-service2
    created
```

Now both applications exist in the *gabor.host* cluster while it is not propagated to the *aks-cluster*.

```
$ kubectl get deployment -n python-ns --context=gabor.host
NAME                 DESIRED   CURRENT   UP-TO-DATE
python-deployment    1         1         1
python-deployment2   1         1         1
    AVAILABLE   AGE
    1           2m
    1           2m

$ kubectl get pod -n python-ns --context=gabor.host
NAME                                  READY     STATUS
python-deployment-55d965fb96-79ddb    1/1       Running
python-deployment2-5f687956c8-hkn82   1/1       Running
    RESTARTS   AGE
    0          2m
    0          2m

$ kubectl get service -n python-ns --context=gabor.host
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP
python-service    ClusterIP   10.233.63.242   <none>
python-service2   NodePort    10.233.46.230   <none>
    PORT(S)          AGE
    5000/TCP         2m
    5001:31112/TCP   2m

$ kubectl get service -n python-ns --context=aks-cluster
No resources found.

$ kubectl get deployment -n python-ns --context=aks-cluster
No resources found.
```

The response to the GET request is successfully received for both applications.

```
$ curl gabor-controller1:31112/5
5. prime number: 11
```

This shows that the application can work together even when deployed as federated resources. To migrate both applications to another cluster, the placement of the namespace needs to be modified just as earlier. No matter how many applications are deployed in such a way, all are moved. In case some of them are supposed to

stay in their current cluster, the placement of the namespace needs to be set to both clusters and the specific resources need to be modified only for migration. Now both applications are moved to *aks-cluster* by setting the placement of the namespace.

```
$ kubectl -n python-ns edit federatednamespace python-ns
federatednamespace.types.federation.k8s.io/python-ns edited
```

When the migration is completed, the resources can be listed.

```
$ kubectl get deployment -n python-ns --context=aks-cluster
NAME                 DESIRED   CURRENT   UP-TO-DATE
python-deployment    1         1         1
python-deployment2   1         1         1
    AVAILABLE   AGE
    1           2m24s
    1           2m24s

$ kubectl get pod -n python-ns --context=aks-cluster
NAME                                  READY     STATUS
python-deployment-ccf59b558-fhmcw     1/1       Running
python-deployment2-85786c748d-tdz56   1/1       Running
    RESTARTS   AGE
    0          2m29s
    0          2m29s

$ kubectl get service -n python-ns --context=aks-cluster
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP
python-service    ClusterIP      10.0.222.197   <none>
python-service2   LoadBalancer   10.0.79.243    40.69.43.88
    PORT(S)           AGE
    5000/TCP          2m27s
    31112:32192/TCP   2m26s
```

In the *gabor.host* cluster, the resources have been terminated.

```
$ kubectl get deployment -n python-ns --context=gabor.host
No resources found.

$ kubectl get service -n python-ns --context=gabor.host
No resources found.
```

Send the GET request to test the application.

```
$ curl 40.69.43.88:31112/15
13. prime number: 41
```

The migration of the applications is successful. The new cluster has the required resources up and running while the old one terminated them.

5 Discussion

Kubernetes Federation v2 is designed to manage multiple clusters by applying resource definitions - YAML files - on the target clusters with a single command. It only requires the kubeconfig files and access to the member clusters. As presented, this can be utilized for migrating stateless applications without redesigning it for the new environment, thus mitigating the problems of vendor lock-in. This approach has its benefits and drawbacks. Some of these are consequences of design decisions while others are due to the fact that Federation was created for multi-cluster management. This chapter compares the presented solution with other approaches. It also describes the key findings of the thesis and discusses some important insights found throughout the work.

5.1 Manual migration

Applications running in Kubernetes can be migrated from one cluster to another manually, without using any extra tool. We can assume a setup similar to the one presented in the thesis and the applications are configured in YAML files.

Migration of applications in this case can take up much time and carries a high risk of human error. The user needs to copy these YAML files to a computer which can communicate with the new cluster. In the case of Federation, this is given due to the fact that the host cluster must be able to communicate with the member clusters. In case there are slight differences in the new environment that the configuration needs to adapt to, the changes need to be carried out manually in the YAML file. This highly increases the risk of human error. Especially if this application is modified in the new cluster then migrated back to the original one and so on. Multiple configuration files need to be kept track of. With more member clusters even more maintenance of multiple files is required. With Kubernetes Federation, a single file is sufficient for multi-cluster or migration purposes. The Federation API can handle overrides for cluster specific configurations. Eliminating the need for multiple configuration files lowers the chance of human error and having multiple versions of an application unintentionally.

Kubernetes Federation v2 utilizes the placement types in the YAML files to decide which clusters a resource needs to be propagated to. As the thesis presented, this can be the name of a cluster but it can also be a label. When the infrastructure includes a couple of clusters only, it is simple to maintain it by names, but with more clusters, it can easily get out of hands. With the use of labels, Federation can target clusters which have a specific label. For instance, it will propagate resources to all the clusters with the label *propapp=true*. Addition or removal of this label on a cluster triggers Federation to act according to the label configurations. In the case of manual migration, it requires the user to take care of this all by hand highly increasing the risk of human error.

5.2 Spinnaker

Spinnaker [74] is an open source software used in cloud environments. It is a continuous delivery platform that can work together with multiple cloud providers. All the major providers are supported, such as Google Compute Engine, Amazon Web Services EC2, OpenStack, Kubernetes, etc [75]. The main purpose of Spinnaker is to make deployment of applications simpler and easier. It is shipped with built-in deployment strategies including canary and red/black. It was created at Netflix and tested on their systems, therefore it is capable of serving enormous infrastructures.

Spinnaker is a mature project compared to Kubernetes Federation v2. The first stable version was released in June, 2017 [76], while Federation is still in alpha state at the time of the thesis work. Hence, fewer bugs and errors can be expected to be present when working with it.

Once Spinnaker is deployed, accounts, also known as deployment targets, can be registered to it. An account contains the necessary credentials needed for authentication [75]. This provides Spinnaker the rights to manage and change the resources. Multiple accounts can be registered in a Spinnaker instance. Therefore, the deployment target can be selected to be any or all of the registered accounts when deploying an application. This is a similar mechanism to the one presented in the thesis and capable of deploying an application in either Kubernetes clusters. Thus, migration of a stateless application is possible using Spinnaker.

Spinnaker is a massive project with a vast amount of features included in it. This comes at a cost. The system requirements for running this project in a cluster is huge in comparison to Federation. The documentation of Spinnaker lists that the minimum requirements are 8 GB memory and a CPU with at least 4 cores [77]. Furthermore, for developing Spinnaker, the memory requirement increases to 18 GB of RAM [78]. These requirements for only application migration purposes are huge. If other features of Spinnaker, not covered by Federation, are expected to be used as well, then it might be worth it to consider this approach to the problem. Federation, in contrast, requires significantly less computer resources. It is not well documented yet, supposedly due to its alpha state. However, rough estimates of the system requirements can be made. The output of the following command describes the configuration of federation controller-manager.

```
$ kubectl get pod -n federation-system -o json
```

The configuration shows a resource limit of 128 MiB of memory and 100m of CPU power as seen on Figure 12. The "100m" value given for the CPU limit means one hundred millicpu. This is the equivalent of the tenth of a single core. Furthermore, the requested resources can be seen as well, showing a minimum of 64 MiB of memory. Table 1 shows these resource requirements and compares Spinnaker to Federation.

Even though the exact system requirements for Kubernetes Federation v2 are not described in the documentation, the upper limits as well as the minimum requirements for the CPU and memory show that this approach requires significantly less

	Kubernetes Federation v2	Spinnaker
Min memory	64 MiB	8 GB
Max memory	128 MiB	n/a
Min CPU	100m	4 cores
Max CPU	100m	n/a

Table 1: Kubernetes Federation v2 and Spinnaker resource requirement comparison

```
"resources": {
    "limits": {
        "cpu": "100m",
        "memory": "128Mi"
    },
    "requests": {
        "cpu": "100m",
        "memory": "64Mi"
    }
},
```

Figure 12: Resource limits and requests for the federation controller-manager pod

computational power than Spinnaker.

Fault tolerance is a greatly important discussion topic within this area. With massive amounts of nodes, deployments and services, failures always have to be assumed to happen. According to Spinnaker's documentation, it is possible to deploy the microservices of Spinnaker independently [77]. This means that the different parts are distributed within a cluster and they are not reliant on each other. If one of them goes down or the containing node goes down, it can be rescheduled on another node without the need to redeploy Spinnaker or reconfigure anything. However, Spinnaker is still deployed in a single cluster and in case the entire cluster goes down, it stops working until it is brought back up. To further increase reliability, these microservices can be deployed in their own clusters as well, which can be in different availability zones. This scenario increases delays and reaction times but considered to be a safer approach. In case a cluster fails or an entire availability zone goes down for some reason, the microservices running in other clusters can still work. Deploying multiple instances of a service, each in their own cluster, possibly in their own availability zones greatly reduces the impact of an outage.

Furthermore, some microservices can be configured to be highly available[79].

This is called sharding, where the service is broke down into smaller pieces. These small shards work similar to the microservices. They are independent of each other in the sense of scaling and configurations. However, this feature is only available for Echo and Clouddriver. Echo is responsible for handling changes, also called webhooks, in services such as Github. Clouddriver takes care of the API calls to the different cloud providers. Spinnaker consists of more than ten microservices, thus this does not cover the whole software but only these two very important ones.

Kubernetes Federation v2 has less features implemented to deal with failures. The controller-manager is deployed in a host cluster and other clusters are joined to the federation as member clusters. In case of an outage where the host cluster goes down, the controller-manager becomes unable to further manage the resources. However, configurations and deployments continue working without interruption as if nothing happened. The only difference is that the federation control plane cannot make changes to the resources it manages until it gets back up. In a general configuration, if a resource has too high load, the controller-manager schedules the creation of more replicas but when the host cluster is down, this could cause an overload, eventually leading to an interruption in the service. In the perspective of migration, no migrations can be executed while the outage lasts. If the outage happens during the process of a migration, it could also result in an interrupted service with no replicas available of a service. If the host cluster runs normally but a member cluster fails, then everything can work fine, since the federation control plane is able to bring up the failed services in any other clusters.

From an availability perspective, Spinnaker definitely performs better. It can be deployed in a distributed way and some of its components can be highly available as well. Meanwhile, Kubernetes Federation v2 stops making changes to the configurations of the managed resources. However, they continue running without interruptions.

5.3 Comments

Kubernetes Federation v2 is a relatively simple, fast and light-weight project that already contains numerous useful features despite it being only a prototype. However, more fault tolerance is required in order to be used reliably in massive systems. Currently, the federation controller-manager can be deployed in a cluster and if either this deployment or this host cluster is down, the federation control plane stops working and no more commands can be executed or configurations can be changed. It would be a good idea to run multiple controller-manager replicas in multiple clusters. This could also be implemented relatively easily. However, since the federation uses the Kubernetes API as a storage for its configurations, problems start to arise because the API is tied to a cluster. Achieving high availability of the federation control plane would essentially mean implementing a highly available Kubernetes API. Then the state of a cluster could be synchronized to a backup storage in real time. No such solution is known at the time of the thesis work.

Even though a highly available federation might never be implemented, disaster recovery could be a more realistic goal. Snapshots could be taken of the federation control plane and in case of a failure, it could be reconstructed in another cluster using the latest snapshot. This solution, though, carries the risk of losing information. One solution could be saving the changes made in the configurations to a storage distributed over multiple clusters. When the host cluster fails, the snapshot and the content of the database would be enough to restore the control plane in another cluster. Furthermore, to make the database reliable and avoid corrupted data, a solution similar to transactions[80] in SQL languages would be required.

Further reducing the effects of a failure in the federation host cluster, the implementation of the propagation could be changed. Currently, the control plane handles the propagation of changes to the member clusters but if the host cluster goes down, the process stops working. Instead, a distributed database should be run over multiple clusters that stores the configurations. The federation control plane writes to this storage, the member clusters read from this storage and apply the changed themselves. However, this solution is likely to introduce extra overhead due to the use of the database resulting in longer delays and higher storage consumption.

5.4 Future Work

In this thesis, the experimental applications are stateless. The presented solutions are true only for this type of applications. In order to continue this work further, stateful applications need to be experimented with as well. From an application migration perspective, the prerequisites for the stateful applications to work are the stateless one. The latter is presented in this thesis. The stateful applications present further challenges, though. For instance, if an application uses a database and stores huge amounts of data in it, migrating it could be difficult. Even if Kubernetes Federation v2 or some other solutions enabled the user to move the database from one cluster to another seamlessly, if the database contains petabytes of data, it would technically be stuck to its original place as moving it would require way too much bandwidth. Distributing databases over clusters might be an approach worth investigating.

6 Conclusions

This thesis introduces the vendor lock-in problem in edge cloud environments. As described, this is a significant problem which needs to be solved. The thesis proposes a tool, Kubernetes Federation v2 as a potential solution for this problem. It is a fast, lightweight and relatively easy-to-use tool designed for multi-cluster management. This thesis shows that is is capable of migrating resources from vendor to vendor. The experimental applications are stateless, thus stateful applications are yet to be tested.

The proposed solution was compared to other, already existing ones. Manual migration quickly becomes complex even in relatively simple infrastructures. Spinnaker is a continuous delivery platform which can be utilized for application migration as well. It is a more mature project than Kubernetes Federation v2 but Spinnaker is significantly more complex with high system requirements. The most significant downside of Kubernetes Federation v2 is that it only works with Kubernetes clusters. However, it is currently the most popular container orchestration tool.

References

[1] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (Jan. 2017), pp. 30–39. ISSN: 0018-9162. DOI: 10.1109/MC.2017.9.

[2] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing*. en. Tech. rep. NIST Special Publication (SP) 800-145. National Institute of Standards and Technology, Sept. 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (visited on 09/22/2019).

[3] Michael Armbrust et al. "A view of cloud computing". en. In: *Communications of the ACM* 53.4 (Apr. 2010), p. 50. ISSN: 00010782. DOI: 10.1145/1721654.1721672. URL: http://portal.acm.org/citation.cfm?doid=1721654.1721672 (visited on 04/09/2019).

[4] W. Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198.

[5] Giovanni Merlino et al. "Enabling Workload Engineering in Edge, Fog, and Cloud Computing through OpenStack-based Middleware". en. In: *ACM Transactions on Internet Technology* 19.2 (Apr. 2019), pp. 1–22. ISSN: 15335399. DOI: 10.1145/3309705. URL: http://dl.acm.org/citation.cfm?doid=3322882.3309705 (visited on 05/13/2019).

[6] *Google Store*. en-GB. URL: https://store.google.com/gb (visited on 05/13/2019).

[7] B. Satzger et al. "Winds of Change: From Vendor Lock-In to the Meta Cloud". In: *IEEE Internet Computing* 17.1 (Jan. 2013), pp. 69–73. ISSN: 1089-7801. DOI: 10.1109/MIC.2013.19.

[8] Karim R. Lakhani and Eric von Hippel. "How Open Source Software Works: "Free" User-to-User Assistance". en. In: *Produktentwicklung mit virtuellen Communities: Kundenwünsche erfahren und Innovationen realisieren*. Ed. by Cornelius Herstatt and Jan G. Sander. Wiesbaden: Gabler Verlag, 2004, pp. 303–339. ISBN: 978-3-322-84540-5. DOI: 10.1007/978-3-322-84540-5_13. URL: https://doi.org/10.1007/978-3-322-84540-5_13 (visited on 05/13/2019).

[9] Haryadi S. Gunawi et al. “Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages”. en. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16*. Santa Clara, CA, USA: ACM Press, 2016, pp. 1–16. ISBN: 978-1-4503-4525-5. DOI: 10.1145/2987550.2987583. URL: http://dl.acm.org/citation.cfm?doid=2987550.2987583 (visited on 05/14/2019).

[10] Saurabh Kumar Garg et al. “Environment-conscious scheduling of HPC applications on distributed Cloud-oriented data centers”. en. In: *Journal of Parallel and Distributed Computing* 71.6 (June 2011), pp. 732–749. ISSN: 07437315. DOI: 10.1016/j.jpdc.2010.04.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731510000936 (visited on 05/14/2019).

[11] Michael J. Kavis. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. en. Google-Books-ID: NcrDAgAAQBAJ. John Wiley & Sons, Jan. 2014. ISBN: 978-1-118-82646-1.

[12] S. Subashini and V. Kavitha. “A survey on security issues in service delivery models of cloud computing”. en. In: *Journal of Network and Computer Applications* 34.1 (Jan. 2011), pp. 1–11. ISSN: 10848045. DOI: 10.1016/j.jnca.2010.07.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804510001281 (visited on 08/13/2019).

[13] Grace Lewis. *Basics About Cloud Computing*. 2010.

[14] Y. Jadeja and K. Modi. “Cloud computing - concepts, architecture and challenges”. In: *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*. Mar. 2012, pp. 877–880. DOI: 10.1109/ICCEET.2012.6203873.

[15] Sumit Goyal. “Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review”. en. In: *International Journal of Computer Network and Information Security* 6.3 (Feb. 2014), pp. 20–29. ISSN: 20749090, 20749104. DOI: 10.5815/ijcnis.2014.03.03. URL: http://www.mecs-press.org/ijcnis/ijcnis-v6-n3/v6n3-3.html (visited on 05/15/2019).

[16] C. Pahl. “Containerization and the PaaS Cloud”. In: *IEEE Cloud Computing* 2.3 (May 2015), pp. 24–31. ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51.

[17] M. Mao and M. Humphrey. “A Performance Study on the VM Startup Time in the Cloud”. In: *2012 IEEE Fifth International Conference on Cloud Computing*. June 2012, pp. 423–430. DOI: 10.1109/CLOUD.2012.103.

[18] Kyoung-Taek Seo et al. “Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud”. en. In: Dec. 2014, pp. 105–111. DOI: 10.14257/astl.2014.66.25. URL: http://onlinepresent.org/proceedings/vol66_2014/25.pdf (visited on 04/12/2019).

[19] A. M. Joy. “Performance comparison between Linux containers and virtual machines”. In: *2015 International Conference on Advances in Computer Engineering and Applications*. Mar. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[20] W. Felter et al. “An updated performance comparison of virtual machines and Linux containers”. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[21] R. Morabito, J. Kjällman, and M. Komu. “Hypervisors vs. Lightweight Virtualization: A Performance Comparison”. In: *2015 IEEE International Conference on Cloud Engineering*. Mar. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[22] *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. URL: http://delivery.acm.org.focus.lib.kth.se/10.1145/2610000/2600241/11600.html?ip=130.237.29.138&id=2600241&acc=ACTIVE%20SERVICE&key=74F7687761D7AE37%2EE53E9A92DC589BF3%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1555508012_c351f12897f3eac024765 (visited on 04/17/2019).

[23] *gnu.org*. en. URL: https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html (visited on 04/15/2019).

[24] *Linux Containers - LXC - Introduction*. URL: https://linuxcontainers.org/lxc/introduction/ (visited on 04/15/2019).

[25] Lee D. McFearin. “Chroot Jail”. en. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 206–207. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_778. URL: https://doi.org/10.1007/978-1-4419-5906-5_778 (visited on 04/15/2019).

[26] Senthil Kumaran S. “Introduction to Linux Containers”. en. In: *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*. Ed. by Senthil Kumaran S. Berkeley, CA: Apress, 2017, pp. 1–9. ISBN: 978-1-4842-3024-4. DOI: 10.1007/978-1-4842-3024-4_1. URL: https://doi.org/10.1007/978-1-4842-3024-4_1 (visited on 04/15/2019).

[27] Rami Rosen. “Linux containers and the future cloud”. In: *Linux J* 240.4 (2014), pp. 86–95.

[28] B. I. Ismail et al. “Evaluation of Docker as Edge computing platform”. In: *2015 IEEE Conference on Open Systems (ICOS)*. Aug. 2015, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

[29] *Enterprise Application Container Platform*. en. URL: https://www.docker.com/ (visited on 04/15/2019).

[30] *Docker 0.9: introducing execution drivers and libcontainer*. en. Mar. 2014. URL: https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/ (visited on 04/15/2019).

[31] *Kata Containers documentation. Contribute to kata-containers/documentation development by creating an account on GitHub*. original-date: 2017-12-21T13:41:45Z. Apr. 2019. URL: https://github.com/kata-containers/documentation (visited on 04/15/2019).

[32] *Google Trends*. URL: https://trends.google.com/trends/explore?date=2013-01-01%202019-08-11&q=%2Fm%2F0wkcjgj,%2Fm%2F0crds9p,Kata%20container (visited on 08/11/2019).

[33] *Machine Types | Compute Engine Documentation*. en. URL: https://cloud.google.com/compute/docs/machine-types (visited on 04/16/2019).

[34] *Amazon EC2 Instance Types for SAP*. en-US. URL: https://aws.amazon.com/sap/instance-types/ (visited on 04/16/2019).

[35] *Eucalyptus*. URL: https://www.eucalyptus.cloud/ (visited on 04/16/2019).

[36] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "Comparison of multiple iaas cloud platform solutions". In: *Proceedings of the 7th WSEAS International Conference on Computer Engineering and Applications,(Milan-CEA 13). ISBN*. 2012, pp. 978–1.

[37] C. Chilipirea et al. "A Comparison of Private Cloud Systems". In: *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. Mar. 2016, pp. 139–143. DOI: 10.1109/WAINA.2016.23.

[38] D. Freet et al. "Open source cloud management platforms and hypervisor technologies: A review and comparison". In: *SoutheastCon 2016*. Mar. 2016, pp. 1–8. DOI: 10.1109/SECON.2016.7506698.

[39] Carlo Daffara. *Comparing OpenNebula and OpenStack: Two Different Views on the Cloud*. en-US. Oct. 2014. URL: https://opennebula.org/comparing-opennebula-and-openstack-two-different-views-on-the-cloud/ (visited on 04/16/2019).

[40] *Apache Cloudstack*. URL: https://cloudstack.apache.org/ (visited on 04/16/2019).

[41] J. P. Mullerikkal and Y. Sastri. "A Comparative Study of OpenStack and CloudStack". In: *2015 Fifth International Conference on Advances in Computing and Communications (ICACC)*. Sept. 2015, pp. 81–84. DOI: 10.1109/ICACC.2015.110.

[42] *Google Trends*. URL: https://trends.google.com/trends/explore?cat=32&date=all&q=%2Fm%2F0cm87w_,Eucalyptus,CloudStack,%2Fm%2F0g58249 (visited on 04/30/2019).

[43] *Get Started, Part 4: Swarms*. en. Apr. 2019. URL: https://docs.docker.com/get-started/part4/ (visited on 04/17/2019).

[44] Lubos Mercl and Jakub Pavlik. "The comparison of container orchestrators". In: *Third International Congress on Information and Communication Technology*. Springer. 2019, pp. 677–685.

[45] *Apache Projects List*. URL: https://projects.apache.org/projects.html?category#big-data (visited on 04/17/2019).

[46] *Apache Mesos*. URL: http://mesos.apache.org/ (visited on 04/17/2019).

[47] *Why Mesos*. en-US. URL: https://mesosphere.com/why-mesos/ (visited on 04/17/2019).

[48] *Google Trends*. URL: https://trends.google.com/trends/explore?cat=32&date=today%205-y&q=Kubernetes,Docker%20Swarm,Apache%20Marathon (visited on 05/22/2019).

[49] *Introducing Container Runtime Interface (CRI) in Kubernetes*. en. URL: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/ (visited on 04/17/2019).

[50] S. Hoque et al. "Towards Container Orchestration in Fog Computing Infrastructures". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. July 2017, pp. 294–299. DOI: 10.1109/COMPSAC.2017.248.

[51] *Building Large Clusters*. en. URL: https://kubernetes.io/docs/setup/cluster-large/ (visited on 04/17/2019).

[52] *Federation*. en. URL: https://kubernetes.io/docs/concepts/cluster-administration/federation/ (visited on 04/30/2019).

[53] *Kubernetes Federation v2. Contribute to kubernetes-sigs/federation-v2 development by creating an account on GitHub*. original-date: 2018-01-09T14:21:03Z. Apr. 2019. URL: https://github.com/kubernetes-sigs/federation-v2 (visited on 04/30/2019).

[54] *Agility through open infrastructure*. en-US. URL: https://www.citynetwork.eu/ (visited on 05/03/2019).

[55] *Ansible deployment of the Kolla containers. Contribute to openstack/kolla-ansible development by creating an account on GitHub*. original-date: 2016-11-15T16:12:46Z. Apr. 2019. URL: https://github.com/openstack/kolla-ansible (visited on 05/03/2019).

[56] *OpenStack Docs: Overview*. URL: https://docs.openstack.org/newton/install-guide-obs/overview.html (visited on 05/06/2019).

[57] *OpenStack Docs: Quick Start*. URL: https://docs.openstack.org/kolla-ansible/latest/user/quickstart.html (visited on 08/11/2019).

[58] *Deploy a Production Ready Kubernetes Cluster. Contribute to kubernetes-sigs/kubespray development by creating an account on GitHub*. original-date: 2015-10-03T20:18:11Z. May 2019. URL: https://github.com/kubernetes-sigs/kubespray (visited on 05/06/2019).

[59] *Kubernetes SIGs*. en. URL: https://github.com/kubernetes-sigs (visited on 05/06/2019).

[60] *Working with Inventory — Ansible Documentation*. URL: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html (visited on 05/06/2019).

[61] *Distributed reliable key-value store for the most critical data of a distributed system: etcd-io/etcd*. original-date: 2013-07-06T21:57:21Z. May 2019. URL: https://github.com/etcd-io/etcd (visited on 05/07/2019).

[62] *Microsoft Azure Cloud Computing Platform & Services*. en. URL: https://azure.microsoft.com/en-us/ (visited on 05/20/2019).

[63] *Overview of kubectl*. en. URL: https://kubernetes.io/docs/reference/kubectl/overview/ (visited on 05/07/2019).

[64] Nikolay Grozev and Rajkumar Buyya. "Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications". en. In: *ACM Transactions on Autonomous and Adaptive Systems* 9.3 (Oct. 2014), pp. 1–21. ISSN: 15564665. DOI: 10.1145/2662112. URL: http://dl.acm.org/citation.cfm?doid=2676689.2662112 (visited on 05/16/2019).

[65] *Redis*. URL: https://redis.io/ (visited on 05/16/2019).

[66] Rajkumar Jalan et al. "Combining stateless and stateful server load balancing". en. US8897154B2. Nov. 2014. URL: https://patents.google.com/patent/US8897154/en (visited on 09/30/2019).

[67] Cesare Pautasso, Erik Wilde, and Rosa Alarcon. "Introduction". en. In: *REST: Advanced Research Topics and Practical Applications*. Ed. by Cesare Pautasso, Erik Wilde, and Rosa Alarcon. New York, NY: Springer New York, 2014, pp. 1–5. ISBN: 978-1-4614-9299-3. DOI: 10.1007/978-1-4614-9299-3_1. URL: https://doi.org/10.1007/978-1-4614-9299-3_1 (visited on 05/08/2019).

[68] *curl*. URL: https://curl.haxx.se/ (visited on 05/08/2019).

[69] *Welcome | Flask (A Python Microframework)*. URL: http://flask.pocoo.org/ (visited on 05/08/2019).

[70] *SymPy*. URL: https://www.sympy.org/en/index.html (visited on 05/08/2019).

[71] *Docker Hub*. URL: https://hub.docker.com/ (visited on 05/08/2019).

[72] *Services*. en. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 05/09/2019).

[73] *Number Theory — SymPy 1.4 documentation*. URL: https://docs.sympy.org/latest/modules/ntheory.html?highlight=prime#sympy.ntheory.generate.prime (visited on 05/20/2019).

[74] *Spinnaker*. en. URL: https://www.spinnaker.io/ (visited on 07/14/2019).

[75] *3. Choose your Environment*. en. URL: https://www.spinnaker.io/setup/install/environment/ (visited on 07/14/2019).

[76] *Versions*. en. URL: https://www.spinnaker.io/community/releases/versions/ (visited on 07/14/2019).

[77] *3. Choose your Environment*. en. URL: https://www.spinnaker.io/setup/install/environment/ (visited on 07/14/2019).

[78] *Getting Set Up for Spinnaker Development*. en. URL: https://www.spinnaker.io/guides/developer/getting-set-up/ (visited on 07/14/2019).

[79] *High Availability*. en. URL: https://www.spinnaker.io/reference/halyard/high-availability/ (visited on 07/31/2019).

[80] rothja. *Transactions (Transact-SQL) - SQL Server*. en-us. URL: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql (visited on 08/01/2019).

A Federation of clusters

```
./go/src/kubernetes-sigs/federation-v2/bin/kubefed2-linux
    join gabor.host --cluster-context gabor.host
    --host-cluster-context gabor.host --add-to-registry --v=2
I0603 09:24:52.227636   21253 join.go:165] Args and flags:
    name gabor.host, host: gabor.host, host-system-namespace:
    federation-system, registry-namespace:
    kube-multicluster-public, kubeconfig: , cluster-context:
    gabor.host, secret-name: , limited-scope: false, dry-run: false
I0603 09:24:52.347296   21253 join.go:219] Performing preflight checks.
I0603 09:24:52.524241   21253 join.go:309] Registering cluster:
    gabor.host with the cluster registry.
I0603 09:24:52.528052   21253 join.go:389] Creating
    cluster registry cluster gabor.host
I0603 09:24:52.538831   21253 join.go:318] Registered cluster:
    gabor.host with the cluster registry.
I0603 09:24:52.538858   21253 join.go:240] Creating
    federation-system namespace in joining cluster
I0603 09:24:52.562380   21253 join.go:248] Created
    federation-system namespace in joining cluster
I0603 09:24:52.562412   21253 join.go:251] Creating
    cluster credentials secret
I0603 09:24:52.562423   21253 join.go:472] Creating
    service account in joining cluster: gabor.host
I0603 09:24:52.652939   21253 join.go:482] Created service
    account: gabor.host-gabor.host in joining cluster: gabor.host
I0603 09:24:52.652979   21253 join.go:510] Creating cluster
    role and binding for service account:
    gabor.host-gabor.host in joining cluster: gabor.host
I0603 09:24:52.710984   21253 join.go:519] Created cluster
    role and binding for service account:
    gabor.host-gabor.host in joining cluster: gabor.host
I0603 09:24:52.711015   21253 join.go:523] Creating secret
    in host cluster: gabor.host
```

```
I0603 09:24:53.747043   21253 join.go:912] Using secret named:
    gabor.host-gabor.host-token-6rd8c
I0603 09:24:53.753755   21253 join.go:944] Created secret
    in host cluster named: gabor.host-tznqt
I0603 09:24:53.753781   21253 join.go:532] Created secret
    in host cluster: gabor.host
I0603 09:24:53.753794   21253 join.go:261] Cluster
    credentials secret created
I0603 09:24:53.753804   21253 join.go:263] Creating
    federated cluster resource
I0603 09:24:53.768093   21253 join.go:272] Created
    federated cluster resource
```

Listing 9: Command and its output when creating a federation of clusters

B Federated python app files

```
apiVersion: v1
kind: Namespace
metadata:
  name: python-ns
```

Listing 10: Namespace definition for the python app

```
apiVersion: types.federation.k8s.io/v1alpha1
kind: FederatedNamespace
metadata:
  name: python-ns
  namespace: python-ns
spec:
  placement:
    clusterNames:
    - gabor.host
```

Listing 11: Federated namespace definition for the python app

```
apiVersion: v1
apiVersion: types.federation.k8s.io/v1alpha1
kind: FederatedDeployment
metadata:
  name: python-deployment
  namespace: python-ns
spec:
  placement:
    clusterNames:
    - gabor.host
    - aks-cluster
```

```
  template:
    metadata:
      labels:
        app: python-app
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: python-app
      template:
        metadata:
          labels:
            app: python-app
        spec:
          containers:
          - image: finta/edgethesispythonapp:latest
            name: python-app
            ports:
            - containerPort: 5000
              protocol: TCP
```

Listing 12: Federated deployment definition for the python app

```
apiVersion: types.federation.k8s.io/v1alpha1
kind: FederatedService
metadata:
  name: python-service
  namespace: python-ns
spec:
  placement:
    clusterNames:
    - gabor.host
    - aks-cluster
  template:
    spec:
      ports:
      - port: 5000
        protocol: TCP
        targetPort: 5000
        nodePort: 31111
      selector:
        app: python-app
      type: NodePort
  overrides:
  - clusterName: aks-cluster
    clusterOverrides:
```

```
    - path: spec.type
      value: LoadBalancer
    - path: spec.ports
      value:
      - protocol: TCP
        port: 31111
        targetPort: 5000
```

Listing 13: Federated service definition for the python app

C Second python application files

```
from flask import Flask
import sympy
import requests

app = Flask(__name__)

@app.route('/<int:number>', methods=['GET'])
def get_prime(number):
  number = int(number)
  if number < 2:
    return "The input needs to be greater or equal to 2\n"
  else:
    address = "http://python-service:5000/" + str(number)
    prime = requests.get(address).text
    prime = int(prime[prime.rfind(" "):].strip())
    retstr = str(prime) + ". prime number: "
    retnum = sympy.prime(prime)
    return retstr + str(retnum) + "\n"

app.run(host='0.0.0.0', debug=False)
```

Listing 14: The Python code for the second application

```
FROM python:3.7-alpine
ADD . /code
WORKDIR /code
RUN pip install flask sympy requests
EXPOSE 5000
CMD ["python", "app2.py"]
```

Listing 15: The Dockerfile for the second application

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: python-deployment2
  namespace: python-ns
  labels:
    app: python-app2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: python-app2
  template:
    metadata:
      labels:
        app: python-app2
    spec:
      containers:
      - name: python-app2
        image: finta/edgethesispythonapp2:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: python-service2
  namespace: python-ns
spec:
  ports:
  - nodePort: 31112
    targetPort: 5000
    port: 5001
    protocol: TCP
  selector:
    app: python-app2
  type: NodePort
```

Listing 16: The YAML file for the second application to run in a Kubernetes cluster

D Federated YAML files for second application

```
apiVersion: v1
apiVersion: types.federation.k8s.io/v1alpha1
kind: FederatedDeployment
metadata:
  name: python-deployment2
  namespace: python-ns
spec:
```

```
  placement:
    clusterNames:
    - aks-cluster
    - gabor.host
  template:
    metadata:
      labels:
        app: python-app2
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: python-app2
      template:
        metadata:
          labels:
            app: python-app2
        spec:
          containers:
          - image: finta/edgethesispythonapp2:latest
            name: python-app2
            ports:
            - containerPort: 5000
              protocol: TCP
```

Listing 17: Federated deployment definition for the second python app

```
apiVersion: types.federation.k8s.io/v1alpha1
kind: FederatedService
metadata:
  name: python-service2
  namespace: python-ns
spec:
  placement:
    clusterNames:
    - aks-cluster
    - gabor.host
  template:
    spec:
      ports:
      - port: 5001
        protocol: TCP
        targetPort: 5000
        nodePort: 31112
      selector:
        app: python-app2
```

```
    type: NodePort
overrides:
- clusterName: aks-cluster
  clusterOverrides:
  - path: spec.type
    value: LoadBalancer
  - path: spec.ports
    value:
    - protocol: TCP
      port: 31112
      targetPort: 5000
```

Listing 18: Federated service definition for the second python app